Sistemas Informáticos
Curso 2006-2007

Soluciones para la gestión de regiones de memoria compartidas por varios hilos

Bruno Contreras Chicote
Miguel Francisco Pérez Villena
Raquel de la Rosa Torres

Dirigido por:
Prof. Teresa Higuera Toledano
Dpto. Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

PALABRAS CLAVE

- Reciclaje de memoria
- Sistemas de Tiempo Real
- Algoritmos de gestión de memoria
- Entorno de programación Java

LICENCIA

Bruno Contreras Chicote, Miguel Francisco Pérez Villena y Raquel de la Rosa Torres autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la memoria desarrollada en este proyecto.

Madrid, a 6 de Julio de 2007

Bruno Contreras Chicote	Miguel Francisco Pérez Villena	Raquel de la Rosa Torres

ÍNDICE

Índice de Figuras

Índice de Algoritmos, Textos y Ejemplos

1.- Resumen del proyecto

La gestión de memoria dinámica es uno de los puntos más importantes dentro de la implementación de java. Una vez que se ha almacenado un objeto en tiempo de ejecución, el sistema hace un seguimiento del estado del objeto, y en el momento en que se detecta que no se va a volver a utilizar ese objeto, el sistema recupera el espacio ocupado de memoria para un uso futuro. Esta gestión de la memoria dinámica hace que la programación en Java sea más fácil, ya que el usuario no se debe preocupar de liberar el espacio ocupado por los objetos (el equivalente de las funciones *dispose* en *Pascal* y *free en C*).

En este proyecto se estudian alternativas a las técnicas clásicas del reciclaje de memoria con el fin de buscar una solución óptima compatible con la ejecución de las aplicaciones de tiempo real crítico.

Proponemos distintos modelos basados todos en el uso del paradigma de una región de memoria adicional introducida por RTSJ (Java para Tiempo Real): la región *Scoped*. Las distintas formas de tratar estas regiones modelizarán las reglas de comportamiento del programa y el modelo de programación, teniendo cada una ciertas ventajas y desventajas aquí analizadas.

Una aproximación a una ejecución bajo modelos propuestos estará simulada mediante una aplicación desarrollada en lenguaje Java, especificando las distintas relaciones entre regiones *Scoped* que se establecen al crear y destruir las regiones implicadas en la ejecución de un programa.

Summary

The performance of any programming language is limited both by the compliance time of programs as his capacity to save memory with the result of house different memory areas that interact on his execution. A basic element here is the garbage collector, which will clear the memory of information not necessary on every very moment.

This project is dedicated to study alternatives of this collector in order that optimize the garbage collection of the programs.

We propose different models based on the use of a paradigm of an additional memory area introduced by RTSJ: the Scoped Memory. The different ways of use of these regions will represent rules behaviour of the program and the programming model, having each one his advantages and inconvenients here analyzed.

We estimate of the execution under the models proposed will be simulated with an application developed on Java, specifying the relationships between Scoped Memory stablished at the creating and deleting the regions implicated on the application execution.

2.- Gestión de memoria en Tiempo Real

El modelo de memoria usado en la Especificación para Java de Tiempo Real *(RTSJ)* incluye un *Heap* con un *Garbage Collector* tradicional, y una nueva administración de la memoria basada en *Scoped Memory* áreas.

Desde las perspectivas de tiempo-real, el recolector de basura introduce pausas impredecibles que no son toleradas por las tareas de tiempo real. Los colectores de tiempo real eliminan este problema, pero introducen un alto sobre coste.

El modelo de *Scoped Memory* áreas asegura a los programadores tiempo constante en sus recuperaciones para de esa manera tener predecibilidad.

Una implementación *RTSJ* fuerza a la validación de una *Scoped* antes de ejecutar una asignación. Ésta validación consistirá en comprobar que un *Memory Area* con un tiempo de vida largo no pueda crear referencias a un objeto ubicado en otro *Memory Area* de tiempo de vida menor. De ahí que *RTSJ* establezca una relación de parentesco entre *Scoped Memory* áreas, que es llamada *Regla del Único Padre*.

Se comprueba esta regla cada vez que un *Scoped Memory* área intenta hacer un ***enter()***, y se explora la pila de *Scoped* cada intento de crear una referencia. Es importante implementar las comprobaciones de las reglas de

asignación de forma eficiente y predecible, ya que las referencias a objetos ocurren frecuentemente.

Sin embargo, esto implica un sobre coste tanto en tiempo de ejecución como en consumo de memoria, lo que hace que el recolector de basura no sea apto para sistemas empotrados de pequeño tamaño. Una alternativa es el uso de regiones de memoria cuya ubicación y desalojo es bajo demanda y mejoran también la localidad espacial. Una facilidad semejante es soportada por *RTSJ*, a través de la introducción de tres tipos de regiones, denominadas *Memory Areas*.

El recolector de basura de forma implícita siempre ha sido reconocido como un beneficioso soporte desde el punto de vista del desarrollo de programas robustos.

2.1.- Gestión de memoria en Java de tiempo real

Tomando como modelo base de la gestión de memoria el libro "*The Real-Time Specification for Java"* sacamos las siguientes conclusiones que sirvieron para empezar a entender la estructura de la máquina virtual de java.

- Permite la definición de regiones de memoria fuera del *Heap* de java.

- Permite la definición de *Scoped Memory* que son regiones con tiempo de vida limitado.

- Permite la definición de regiones de memoria con objetos inmortales cuyo tiempo de vida es el de la aplicación

- Permite la definición de regiones de memoria que soportan características físicas.

- Permite la especificación de los tamaños máximos de memoria y los tipos de la memoria para los hilos de tiempo real.

- Permite al programador observar el comportamiento del recolector y sus límites

Las siguientes listas establecen los requisitos y los valores semánticos que serán aplicables en las clases (constructores, métodos y campos) referidas a la gestión de memoria:

1. Un área de memoria *Scoped* está definida por una referencia a la clase *ScopedMemory*. Cuando una *Scoped* entra mediante el método ***enter***() o iniciando una llamada a *RealTimeThread* o *NoHeapRealTimeThread* cuyos constructores hicieron una referencia a *ScopedMemory*, todos los siguientes

usos de la palabra clave (la referencia) dentro del programa ubicará la memoria desde una referencia a la *ScopedMemory*.
Cuando el *Scoped* sale con el retorno de ***enter***() a su referencia a *ScopedMemory* ubicará la memoria desde su área al cierre del *Scoped.*

2. Cada instancia de *ScopedMemory* o de sus subclases debe tener un ***contador de referencias*** del número de *Scopeds* en los que está siendo usado.

3. El contador de referencias de *ScopedMemory* o una de sus subclases es incrementado en uno cada vez que hay una referencia a la instancia en el constructor de un *RealTimeThread* o *NoHeapRealTimeThread.*

4. El contador de referencias de *ScopedMemory* se decrementa en uno cuando se referencia la vuelta un método ***enter***(), cuando una instancia de *RealTimeThread* o *NoHeapRealTimeThread* y el área asociada es mediante una referencia en el hilo *MemoryParameters*. El otro caso es que un *Scoped* interno retorne de su método ***enter***().

5. Cuando el contador de referencias de *ScopedMemory* o una de sus subclases pasa de uno a cero todos los objetos involucrados dentro son consideras inalcanzables y puestos como candidatos pendientes de reclamación. Los métodos finalizadotes de cada objeto de memoria involucrados en el *ScopedMemory* en cuestión son ejecutados por completo antes de cualquier otro intento para coger memoria por parte de otro hilo.

6. Los *Scoped* pueden estar anidados. Cuando un *Scoped* es anidado todas las siguientes referencias de memoria serán asociadas a este nuevo *Scoped*. Cuando un *Scoped* anidado sale las subsecuencias pasarán de nuevo al anterior.

7. Los *MemoryArea* asociados con un *NoHeapRealTimeThread* no podrán mover ningún objeto.

8. Los objetos creados en cualquier zona de memoria inmortal tienen el mismo tiempo de vida que la aplicación. Terminará cuando actúen los métodos finalizadores de la aplicación.

9. Cada instancia de la máquina virtual tendrá una instancia de la *InmortalMemory*.

10. Cada instancia de la máquina virtual tendrá una instancia de la *HeapMemory*.

11. Existen restricciones para proteger los punteros creados en java, siendo las siguientes (Figura 1):

	Referencia a Heap	Referencia a Inmortal	Referencia a Scoped
Heap	Si	Si	No
Inmoral	Si	Si	No
Scoped	Si	Si	Sí, si es el mismo está fuera o es scope compartido
Variable Local	Si	Si	Sí, si es el mismo está fuera o es scope compartido

Figura 1. Restricciones

12. Cualquier implementación de la máquina virtual *RTSJ* debe asegurarse de que las comprobaciones superiores se ejecutan antes de que se ejecute la declaración.

Los lenguajes que emplean una demanda automática de bloques de memoria usan normalmente el algoritmo denominado recolector de basura. Los algoritmos de recolección de basura varían en la cantidad de antidependencia que añaden a la ejecución de la lógica de programa. Actualmente se cree que algoritmos o implementaciones que no sean las de recolectores de basura permitan actuar en los espacios dejados entre los punteros a los objetos en el *Heap* de una forma consistente y que son suficientemente cercanos en tiempo para minimizar el elevado tiempo de latencia en el intercambio de datos entre tareas de tal forma que puedan ser considerados apropiados para todos los sistemas de tiempo real.

Por todo esto la especificación aquí planteada proporciona el acercamiento a las áreas de memoria permitiendo a los programas lógicos alojar objetos en una especie de Java, ignorar al reclamación de objetos y no incurrir en al latencia de la implementación de los algoritmos de recolección de basura.

2.2.- Especificación para tiempo real de Java

2.2.1.- Memory Area

Clase de memoria en la que se pueden almacenar objetos, cuando se crea se especificará su tamaño. Es la clase abstracta base de todas las clases de acuerdo con representaciones de asignación de áreas de memoria, incluyendo el área de memoria inmortal, la memoria física y los áreas de la *Scoped Memory*.

Constructor

protected **MemoryArea**(long sizeInBytes)

Métodos

public void **enter**(java.lang.Runnable logic)

Asocia este área de memoria al hilo de tiempo real en curso durante la ejecución del método ***run***() pasado como parámetro. Durante ese periodo de ejecución vinculado, todos los objetos son asignados desde el área de memoria hasta otro que tenga efecto, o el método ***enter***() termine. Una excepción de tiempo real es lanzada si este método es llamado desde otro hilo como un *RealtimeThead* o *NoHeapRealtimeThread*.

public static MemoryArea **getMemoryArea**(java.leng.Object object)

Devuelve el área de Memoria en la que el objeto está alojado.

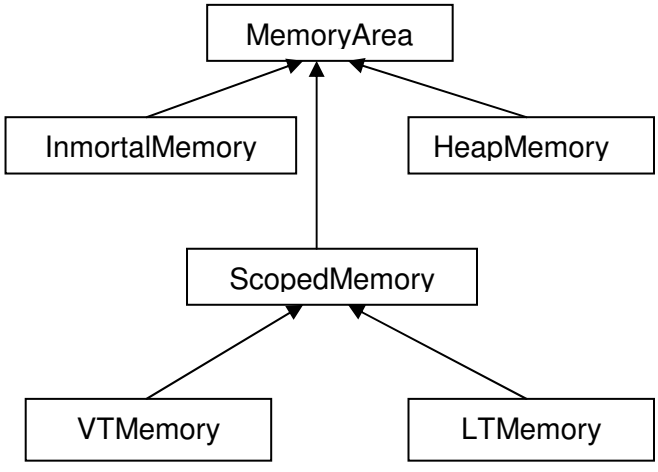


Figura 2. Estructura de la MemoryArea

2.2.2.- Heap Memory

La clase *HeapMemory* es un objeto *singleton* que permite lógica dentro de otros *Scoped Memory* para vincular objetos en el *Heap* de Java. El *Heap* es la zona de memoria para la gestión de las variables que permanecen un largo tiempo, las cuales siguen existiendo aún cuando la función la haya terminado. Se pide mediante *malloc*(), *brk*() y se librera con *free*().

2.2.3.- InmortalMemory

La Memoria Inmortal es un recurso compartido por todos los hilos. Los objetos vinculados en la memoria inmortal viven hasta el fin de la aplicación. Objetos en la memoria inmortal nunca son objetos del recolector de basura, aunque algunos algoritmos de GC deben requerir un escáner de la memoria inmortal. Una memoria inmortal solo debe contener una referencia a objetos de otra memoria inmortal o a objetos del *Heap*.

2.2.4.- ScopedMemory

Es la clase abstracta que trata con las representaciones de espacio de memoria con un tiempo de vida limitado. El área *Scoped Memory* es válido mientras tenga hilos que tengan acceso a él. Una referencia es creada por cada método que intenta acceder cuando cualquier hilo de tiempo real es creado con *Scoped Memory* y su área de memoria, o un hilo de tiempo real ejecuta el método public void **enter** (*java.lang.Runnable logic*), los terminadores se ejecutan para todos los objetos del área de memoria siendo el área vaciado. Un área *Scoped Memory* es una conexión a un área particular de memoria indicando el estado actual de éste. El objeto no posee necesariamente referencias directas a la región de memoria de la cual es dependiente en la implementación

Cuando un área de memoria *ScopedMemory* es instanciado, el objeto es destinado automáticamente al área de memoria en uso en ese momento, pero que no es el espacio de memoria que representa ese objeto. Lo normal es que el área de memoria para la *ScopedMemory* pueda ser referenciado por los típicos métodos que implementan el ***malloc****()* y ***free****()*, o rutinas similares que modifican el área de memoria. El método ***enter***() es el usado para activar una nueva memoria *Scoped*. Entrar en la memoria *Scoped* se realiza mediante: public void ***enter***(*Runnable* r) siendo r un objeto de tipo *Runnable* cuyo método ***run***() representa el punto de entrada al código que ejecutará el nuevo *Scoped*. La salida de la región S*coped* ocurre cuando el r.***run***() se completa. Cuando r.***run***() termina el área de la memoria *Scoped* deja de ampliarse. Su contador de referencias de decrementará y si pasa a cero todos los objetos de la memoria finalizarán y serán recogidos.

Los objetos referenciados desde el área de la *Scoped Memory* tienen un tiempo de vida único. Dejan de existir con el método ***enter****(java.lang.Runnable logic)* o teniendo referencias superiores del último hilo de tiempo real, a pesar de que haya alguna referencia más al objeto. De este modo, para mantener la

seguridad de Java y permitir referencias pendientes existen una serie de reglas restrictivas de aplicación sobre el área del *Scoped Memory*:

a. Una referencia a un objeto en la *Scoped Memory* nunca puede ser almacenado en un objeto alojado en el *Heap.*

b. Una referencia a un objeto en la *Scoped Memory* nunca puede ser almacenado en un objeto alojado en la *Inmortal Memory.*

c. Una referencia a un objeto en la *Scoped Memory* sólo puede ser almacenado en un objeto de la misma *Scoped Memory* o en una *Scoped Memory* más interna habiendo sido introducida por el método **enter***().*

d. Referencias a objetos de *Inmortal Memory* o *Heap* deben ser almacenados dentro de un objeto alojado en el área de la *Scoped Memory.*

Constructor

public **ScopedMemory**(long sizeInBytes)

Métodos

public void **enter**(java.lang.Runnable logic)

Asocia esta *Scoped Memory* al hilo de tiempo real en uso por el tiempo que dure la ejecución del método ***run***(). Durante este periodo todos los objetos son destinados desde el área de la *Scoped Memory* hasta que otro toma efecto, o exista el método **enter***().* Una excepción de ejecución saltará si este método es llamado desde un hilo distinto de *RealtimeThead* o *NoHeapRealtimeThead.*

public int **getMaximunSize**()

public MemoryArea **getOuterScope**()

public java.lang.Object **getPortal**()

public void **setPortal**(java.lang.Object object)

Este es el tipo en el que centraremos nuestros esfuerzos. Heredarán de él: *VTMemory* cuyo tiempo de reserva de espacio es variable y *LTMemory* que garantiza un tiempo lineal (dependiente del tamaño del objeto) de reserva de espacio para los objetos que se creen en ella aunque no se garantiza que esté disponible el tamaño máximo cuando se necesita.

VTMemory

El tiempo de ejecución de una referencia desde una *VTMemory* debe tomar una cantidad de tiempo variable. Estas áreas pueden ser usadas por instancias de hilos de *NoHeapRealtimeThead*.

Constructor

public **VTMemory**(int inicial, int maximun)

LTMemory

Representa un área de memoria, asociado por *RealTimeThead*, o por un grupo de hilos de tiempo real, estando garantizado por el sistema que tendrá un tiempo de asignación lineal. El área de memoria descrita por una instancia de *VTMemory* no existe en el *Heap* de Java, y no es objeto del *Garbage Collector*. Un área *VTMemory* tiene un tamaño inicial, siendo el tamaño de memoria suficiente para los requerimientos del constructor. La memoria inicial debe comportarse, para lograr un alojamiento idóneo, como si fuera continua, por ejemplo una correcta implementación debe garantizar que cualquier secuencia de objetos puedan ser asociados en algún momento correctamente sin exceder una tamaño de memoria inicial. El tiempo de ejecución a la hora de vincular un objeto desde este área de memoria inicial debe ser lineal respecto al tamaño del objeto vinculado.

Constructor

public **LTMemory**(long initialSizeInBytes, longmaxSizeInBytes)

Uso del ScopedMemory

Es difícil escribir código Java sin crear objetos temporales. El caso extremo lo podemos ver en los objetos permanentes, como las instancias de *String*, *Integer* o *BigInteger*. No se puede concatenar o crear sub-cadenas de *String* sin crear nuevos objetos *String*.

Las técnicas de reciclaje no funcionan adecuadamente con los objetos permanentes. Por lo que, si se quiere evitar el uso del *Heap,* una de dos: será necesario trabajar sin algunas de las facilidades del entorno Java, o se necesitará una forma de evitar que estos objetos saturen la memoria inmortal.

Este mecanismo es el *Scoped Memory*. Pero su uso puede ser complicado. Para llevar a cabo esta sección nos hemos basado en [9]

Memoria Temporal

Consideremos el siguiente ejemplo: Se tiene un *String* que representa un valor entero y se quiere pasar a un valor *Integer*. Una forma para hacer la transformación sería escribir el código de conversión sin consumir objetos, pero usando las herramientas de la clase *Integer,* lo que además ahorraría tiempo de programación.

int valor = Integer.valueOf(str).intValue();

En código Java, esta instrucción es correcta, el método *valueOf* devuelve un objeto *Integer* que ya no se usa después de la instrucción y es recogido por el *Garbage Collector*.

En código *RTSJ*, sin usar el *Heap*, el valor intermedio *Integer* supone un gasto de memoria, ya que si no se utiliza el *Heap* se almacena en *Inmortal* (a no ser que se utilice *Scoped*). La solución está en meter la conversión en un área de *Scoped Memory*:

```
Runnable convertidor = new Runnable()
  {
    public void run()
      {
        valor = Integer.valueOf(str).intValue();
      }
  }
scopedMemory.enter(convertidor)
```

Algoritmo 1. Convertidor

El valor intermedio *Integer* se crea en el *Scoped Memory* y se recicla al retornar el método ***enter()***.

Pero hay un problema, esto previene el agujero de memoria producido por *Integer* pero crea otro agujero de memoria, el generado por el objeto convertidor. El método ***enter()*** requiere el objeto convertidor, y este no puede ser alojado en el mismo ámbito en el que está *Integer*.

Si la conversión de *String* a *Inegert* ocurre una sola vez en la ejecución de la aplicación el agujero producido por *Integer* (o *Runnable*) está acotado. Esto no supone un problema, y ni siquiera merece la pena ser atacado con *Scoped Memory*. El verdadero problema aparece cuando la conversión se produce en un bucle:

```
while (done == false)
  {
    String linea = getIntegerString();
    int valor = Integer.valueOf(linea).intValue();
    ...
  }
```

Algoritmo 2. Bucle

Este código creará un agujero de memoria por cada *String* e *Integer* en cada turno.

```
Runnable convertidor = new Runnable()
  {
    public void run()
      {
        String linea = getIntegerString();
        valor = Integer.valueOf(linea).intValue();
        ...
      }
  }

while (done == false)
  {
    scopedMemory.enter(convertidor);
  }
```

Algoritmo 3. Agujero String

El código crea una instancia única de *Runnable* (convertidor) en el primer ámbito del contexto (llamado memoria *Inmortal* en el *Heap*), y luego crea un sin fín de objetos *Integer* y *String* en el área de *Scoped Memory*. Estos objetos son liberados cuando dejan el *Scoped Memory*.

Cuando *Scoped Memory* se usa como almacenamiento temporal, lo primero que preocupa al programador son las consecuencias de las reglas de asignación:

1. ***Seguramente no se pueda almacenar ningún objeto temporal en un objeto que no se haya creado en el método run.***

No hay que tener en cuenta los objetos creados en métodos llamados desde *run*, pero los objetos creados antes de invocar el método sí que pueden ser un problema. Hay formas de trabajar en torno a esto, pero es mejor evitar cada situación usando técnicas como las siguientes:

- Aumentar la cantidad de código que usa el *Scoped*, hasta que abarque todos los objetos en los que necesita almacenar referencias.
- Cambiar las definiciones de los objetos para que los valores que almaceno en objetos creados fuera del "alcance temporal" sean valores primitivos y no referencias a objetos.

2. *¿Cómo voy a coger los resultados fuera de la región?*

Este problema es real, pero hay patrones de código que ayudan a mover los resultados fuera del *Scoped Memory*.

Patrones de Código para Memoria Temporal

***Parámetros de permiso*.**

Las referencias que atraviesan la frontera del *Scoped Memory* suelen causar problemas. Este patrón pasa los valores a un cómputo en una región temporal, y previene el problema limitando el pase de parámetros a la región temporal como:

- Valores primitivos.
- Objetos permanentes.
- Objetos con vida limitada que son considerados permanentes dentro del *Scoped*.

Este patrón de código tiene las siguientes partes:

1. Define la clase *Runnable*, X, con un campo para cada parámetro y un constructor que copia esos valores desde los argumentos.

2. Define el método *run*, que lleva a cabo el trabajo que debería hacerse en la región temporal.

3. Construye una instancia de X usando un constructor que pasa en los parámetros.

4. Ejecuta la instancia de X con *enter(x).*

Es decir, este patrón crea una clase, la cual tendrá un constructor al que le pasaremos los atributos que vaya a utilizar por parámetro. Y creará una copia interna de estos valores. Cuando creemos un objeto de la clase podremos meterla en *Scoped Memory*, de tal forma que todo lo que use será eliminado al salir del *Scoped*, y no será un problema porque esas variables serán copias de los valores originales.

```
Class PasarPorRef implements Runnable {
  String nombreArchivo;

  PasarPorRef(String nombre) {nombreArchivo = nombre;}

  public void run() {
       try {
         File archivo = new File(nombreArchivo);
         System.out.println ("La ruta del archivo es "
         + archivo.getAbsolutePath());
           }

       catch (Throwable th) {
        System.err.println("Excepción no capturada en Scope. " + th);
       throw new RuntimeException(th);
         }
       }
   }

PasarPorRef pasarPorRef = new PasarPorRef(".");
scopedMemory.enter(pasarPorRef);
```

Ejemplo 1. Pasar un valor al Scope

En el ejemplo 1 la clase *Runnable* llamada X en la descripción es *PasarPorRef*. El parámetro es un valor *String* (*String* es una clase permanente). Las dos últimas líneas del ejemplo demuestran los pasos 3 y 4 creando una instancia de *PasarPorRef* e invocándola en el *Scoped Memory*.

Será fácil leer el código que pasa parámetros de manera invisible. Las clases internas pueden usar valores finales en el código circundante, por lo que nosotros podemos pasar valores haciéndolos finales, así:

```
Final String nombreArchivo = ".";
Runnable runnable = new Runnable() {
  Public void run() {
      try {
       File archivo = new File(nombreArchivo);
       System.out.println("La ruta del archivo es
       "+archivo.getAbsolutePath());
          }
      catch (Throwable th){
       System.err.println ("Excepción no capturada en Scope. " + th);
       throw new RuntimeException(th);
         }
       }
   };
scopedMemory.enter(runnable);
```

Ejemplo 2. Pasar valores al Scoped usando variables finales

Este patrón pasa valores al cómputo del *Scoped* de manera más clara que el patrón mostrado en el Ejemplo 1, pero pasar valores a través de un constructor hace que el mecanismo sea explícito. Es una elección entre el equilibrio de la claridad añadida por el código corto y la claridad añadida por pasar parámetros explícitamente.

No hay nada mágico detrás de los valores finales. Si se puede, el compilador sustituye las constantes por variables finales. Los valores que el compilador no puede evaluar pasan a ser campos ocultos en la clase interna cuando el objeto es creado.

Retorno de valores primitivos

Las reglas de asignación del *RTSJ* no se aplican a los valores primitivos, por lo que los valores primitivos pueden ser devueltos desde *Scoped Memory* tan fácilmente como son retornados desde la memoria inmortal. Los pasos son:

1. Define la clase *Runnable*, X, con una instancia de campo con tipo primitivo, y, para su valor de retorno.
2. Define el método *run* que lleva a cabo el trabajo que debería hacerse en la región *Scoped*.
3. Construye una instancia de X.
4. Ejecuta la instancia de X con *enter(x).*

```
Class GranCalculo implements Runnable {
  long resultado;
  private String fUno;
  private String fDos;

  GranCalculo(String f1, String f2) {
    fUno = f1;
    fDos = f2;
   }

  public void run() {
    try {
      GranCalculo f1 = new BigInteger(fUno);
       GranCalculo f2 = new BigInteger(fDos);
       GranCalculo f3 = f1.multiply(f1);
       GranCalculo f4 = f3.divide(f2);
       resultado = f4.longValue();
    } catch (Exception ex) {
          throw new RuntimeException (ex.toString());
       }
  }
}
GranCalculo convertidor = new GranCalculo ("123456789123456789",
                                    "98765432198765432131415 9");
scopedMemory.enter(convertidor);
System.out.println("El resultado es" + convertidor.resultado);
```

Ejemplo 3. Devolver un valor primitivo desde el cómputo del scope

5. Recupera el valor de retorno desde y.

En el ejemplo 3 *GranCalculo* declara una instancia visible de la variable resultado donde deja un valor *long* para que el método invocador pueda hacer uso de él. Este ejemplo también utiliza la técnica del Ejemplo 1 para pasar dos argumentos al cálculo del *Scoped*.

Un patrón alternativo pasa el valor primitivo devuelto a través del campo en una clase circundante.

```
final String fUno = "12345678912345678 9";
final String fDos = "98765432198765432131 4159";

Runnable runnable = new Runnable() {
  Public void run() {
    try {
       BigInteger f1 = new BigInteger(fUno);
       BigInteger f2 = new BigInteger(fDos);
       BigInteger f3 = f1.multiply(f1);
       BigInteger f4 = f3.divide(f2);
       Outer.this.resultado = f4.longValue();
      } catch (Exception ex) {
           throw new RuntimeException (ex.toString());
      }
   }
};

scopedMemory.enter(convertidor);
System.out.println("El resultado es" + convertidor.resultado);
```

Ejemplo 4. Una manera alternativa de devolver un valor primitivo

Este patrón necesita cooperar con la clase circundante (creando y reservando el campo del valor de retorno), pero apoya las clases internas anónimas.

Este capítulo evita el patrón *scopedMemory.enter(new Runnable()).*

Desde que hacemos uso de la *Scoped Memory* para evitar la creación de objetos temporales, es imposible que una creación casual de Runnables temporales sea aceptable.

Referencias al objeto de retorno

Puede ser difícil devolver objetos desde *Scoped Memory*. Los objetos creados en él serán reciclados cuando el control los devuelva a la función padre, y las reglas de asignación no permitirán que la aplicación ponga referencias a los objetos que el padre pudiese ver. Para devolver un objeto desde *Scoped Memory*, habrá que ubicar el objeto de retorno en una memoria contextual donde la función padre pueda verlo.

Los pasos de este patrón de código son:

1. Definir una clase *runnable* que incluya un campo, por ejemplo resultado, del tipo apropiado.

2. En el constructor, inicializar *resultado* con una instancia del valor de la clase de retorno.

3. En el método *run,* guardar los valores primitivos deseados en *resultado*. Solo los valores primitivos pueden llevarse al *Scoped* temporal, por lo que si el tipo de retorno debe incluir referencias a objetos, estos objetos tienen que crearse en el constructor.

4. Crear una instancia de la clase del primer paso, y entrar en ella.

5. Cuando la función *enter()* termine, recupera el valor de retorno desde el campo return.

```
class GranCalculo implements Runnable{
  char [] resultado;
  private String fUno;
  private String fDos;

  GranCalculo(String f1, String f2, int longitudMax({
    fUno = f1;
    fDos = f2;
    resultado = new char[longitudMax];
  }

  public void run() {
    try {
      GranCalculo f1 = new GranCalculo (fUno);
      GranCalculo f2 = new GranCalculo (fDos);
      GranCalculo f3 = f1.multiply(f1);
      GranCalculo f4 = f3.divide(f2);
      String s = f4.toString();
      if (s.length() > resultado.length) {
        throw new RuntimeException( "BigCompute2 necesita un
resultado seleccionado al   final " + s.length() + "longitud.");
      }
      for (int i = 0; i<s.length(); ++i){
        resultado[i] = s.charAt(i);
```

```
      }
      for (int i = s.length(); i<resultado.length; ++i) {
        resultado[i] = ' ';
      }
    } catch (Exception ex) {
        throw new RuntimeException (ex.toString());
      }
    }
  }
}

final int LONGITUD_MAX_RESULT = 40;
BigCompute conversor = new BigCompute("12345678912345678 9",
                            "12345", LONGITUD_MAX_RESULT);
scopedMemory.enter(conversor);
System.out.print("El resultado del cálculo es ");
for (int i = 0;
     i < LONGITUD_MAX_RESULT && conversor.resultado[i] != ' ';
++i) {
  System.out.print(conversor.resultado[i]);
}
System.out.println();
```

Ejemplo 5. Retorno de un valor de objeto desde el cómputo del scoped

Las reglas de asignación impiden la creación de punteros colgantes. No pueden crearse referencias a objetos creados en los patrones de memoria temporal desde variables estáticas, ni siquiera en el objeto *Runnable* definido en el paso 1 de cada patrón.

Resumen de las reglas para la memoria temporal

- Los valores que se pasan al interior deben ser datos primitivos u objetos permanentes.

- Los valores que se pasan al exterior deben ser datos primitivos u objetos que pueden ser inicializados completamente con valores primitivos, usando métodos *set*.

- Las referencias a objetos nuevos no pueden ser almacenados en objetos estáticamente accesibles, incluso el objeto del hilo actual.

- Las referencias a objetos nuevos no pueden almacenarse en los patrones del objeto *Runnable*.

Figura 3. Resumen reglas de memoria temporal

Encapsulado de hilos de tiempo real en Scoped Memory

Si un hilo de tiempo-real se ejecuta completamente en una región de *Scoped Memory* que no es compartido con otras tareas, es decir, solo lo tiene el hilo de tiempo real, ese *Scoped Memory* constituye un contexto de asignación privado para ese hilo. El hilo de tiempo real puede hacer algo con los objetos que crea, y estos son recolectados cuando el hilo termina. Aquí, "algo" significa:

1. Las referencias a los objetos creados por el hilo no pueden ser almacenados ni en el *Heap* ni en la memoria inmortal.

2. Es complicado pasar referencias entre el hilo que está en la región *Scoped Memory* y los que están en otras áreas de memoria.

El patrón de *hilo encapsulado* trabaja aislando un objeto programado en *Scoped Memory*. Si ve algo fuera del *Scoped Memory*, es a través de una mirilla estrecha.

La diferencia principal entre un hilo encapsulado, y el uso ordinario del *Scoped Memory*, es que todo aquello que pertenece al entorno del hilo encapsulado está en un *Scoped Memory* único.

El siguiente trozo de código muestra el uso de *Encapsulate* desde el *Heap* o la memoria *Inmortal*.

```
Encapsulated ti = new Encapsulated(64*1024);
ti.encapsulated(RunThis.class);
try {
  ti.waitForCompletion();
  }
  catch (InterruptedException ie) {
    // Ignore
}
```

Ejemplo 6. Forma larga de usar el encapsulado

El siguiente ejemplo muestra un ejemplo de código que puede ser lanzado desde *Encapsulated*. Lo más significativo del ejemplo es lo simple que es.

```
static class RunThis implements Runnable {
  public void run() {
    final String fNombre = “TempImmortal.java”;

    try {
      File archivo = new File(fNombre);
      BufferedReader entrada = new BufferedReader(
        New FileReader(archivo));

      String linea;
      while ((linea = entrada.readLine()) != null) {
        System.out.println(linea);
      }
      input.close();
    } catch (java.io.IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Ejemplo 7. Código que ejecuta en un hilo encapsulado

La memoria requerida del ejemplo es notable. Usa bastante *Scoped Memory* porque requiere memoria proporcional al tamaño de la copia.

El próximo ejemplo ilustra una manera de invocar *encapsulate*, pero nótese que ejecuta el constructor *Encapsulate* por cada llamada de encapsulate.

```
new Encapsulated (20*1024).encapsulate(RunThis2.class);
```

Ejemplo 8. Forma-breve del uso del encapsulado

La clase *Encapsulate* implementa la envoltura que da comienzo al hilo encapsulado. Su API expone un constructor y dos métodos:

`encapsulate(long size)`: Crea una instancia de `Encapsulate` usando una `VTMemory` del tamaño especificado.

`encapsulate(Class logicClass)`: Encapsula el método `run` desde una instancia de `logicClass`, que debe ser `Runnable`.

`waitForCompletion()`: Espera a que termine el hilo encapsulado.

Clases Internas

Las clases internas no-estáticas pueden referenciar campos de instancias en la instancia que las ha creado, y las clases internas locales también pueden usar variables locales finales y parámetros en los métodos.

El programador de Java puede disfrutar de las facilidades de las clases internas sin preocuparse por el mecanismo subyacente (interno), pero *Scoped Memory* trae esos mecanismos ocultos a primer plano, forzando a los programadores de *RTSJ* a pensar sobre el mecanismo que soporta a las clases internas.

El compilador añade parámetros ocultos a los constructores de las clases internas. Pasa todos los valores externos que necesitará, como:

Outer this: El valor de la clase en el momento en que es construida la instancia de la clase interna.

Final values: El valor de cada valor final que es:

- Visible cuando la instancia interna es construida.
- Usado en la clase interna.
- Asignado su valor de tal forma que el compilador no pueda calcular su valor constante y usar esa constante en la clase interna.

Esta conducta de las clases internas es inofensiva para las clases internas usadas con *enter* en *Scoped Memory*, pero a menudo se producen errores en *executeInArea* cuando se usa desde *Scoped Memory*. La regla básica es:

No usar una clase interna con `executeInArea` desde un área de *Scoped Memory*

Para entender el problema, considerar:

```
ImmortalMemory.instance().executeInArea(new Runnable(){
  Public void run() {
    System.out.println("Hola");
  }
});
```

Esto parece inofensivo, pero el constructor oculto sería algo como:

```
Outer$1$Inner(v1) {
  This$0 = v1;
}
```

Si el puntero a *this* para la función llamante indica un objeto en *Scoped Memory*, el constructor cogería un *IllegalAssignmentError* cuando tratase de almacenarlo en *this$0*.

No siempre es tan fácil

Este capítulo se ha restringido a las aplicaciones del *Scoped Memory* que no se involucran compartiendo esas áreas de memoria.

- *Scoped Memory* puede ser compartido. En cuanto un hilo de tiempo-real se crea y comienza en un *Scoped*, el nuevo hilo comparte el *Scoped* con el hilo que lo creó.

- Y, el *RTSJ* permite a las aplicaciones "ver" la pila de las áreas de memoria en las que han entrado y heredado de hilos padre.

Los objetos en *Scoped Memory* son finalizados cuando el *Scoped* esta vacío. Esto raramente es un problema pero:

1. Los finalizadores pueden "resucitar" al *Scoped* que contiene el objeto creando un nuevo contexto de ejecución en ese *Scoped*. Un aspecto especialmente difícil de esta conducta es que los finalizadotes solamente se ejecutan una vez, por lo que aunque los objetos del *Scoped* puedan ser usados hasta que su *Scoped* esté limpio, solo serán finalizados la primera vez que el *Scoped* que los contiene llega a estar sin referenciarse desde ningún lado

2. Los finalizadores se ejecutan mientras *enter* está retornando. Si los finalizadores son costosos, retornar desde *enter* podría llevar mucho tiempo incluso, la finalización podría ser infinita si un finalizador tiene un bucle infinito.

Evitar los finalizadores para objetos en **Scoped Memory**

La finalización no debería ser la primera elección del programador.

Implementación del Encapsulado

El uso de *Encapsulate* tiene 6 pasos:

1. El constructor crea un área privada de *Scoped Memory* en el *Heap*, y lo guarda en *target*.

2. *encapsulate* usa *executeInArea* para dar a la función llamante una pila del *Scoped* que contiene solo memoria del *Heap*.

3. Crea una instancia de *Step3* en el *Heap*, y entonces usa *target.joinAndEnter(step3)* para:

   - Esperar hasta que *target* no esté en uso.
   - Entrar en *target* e invocar el método *run* de *Step3*.

4. El método *run* de *Step3* deberá ejecutarse en *Scoped Memory*. Ahí, crea un hilo de tiempo-real configurado para ejecutar una instancia de *logicClass* construido en el constructor de *Step3*.

5. Comienza el hilo creado en el paso anterior. Este es el *Hilo Encapsulado*. Este hilo empezará con una pila de *Scoped* conteniendo solamente *Scoped Memory* llamado *target* y memoria del *Heap*.

6. Si la función llamante quiere esperar al hilo encapsulado para terminar, puede llamar a *waitForCompletion,* que usa el método *join* del *Scoped Memory* para retardar hasta que ningún hilo esté usando *target*.

```
static class Encapsulate {
  private ScopedMemory target;

  static class Step3 implements Runnable {
    Runnable logico;
    ScopedMemory cargador;

    Step3 (Class logicClass, ScopedMemory ma) {
      target = ma;
      try {
        logico = (Runnable)logicClass.newInstance();
      } catch (Exception ex) {
           // Can't throw usefully, so just
           // throw a runtime exception
           throw new RuntimeException(ex.toString());
        }
    }
```

```
    public void run() {
      RealtimeThread step4 = new RealtimeThread(
                null, null, null, null, null, logico);
      step4.start();
    }
  }

  Encapsulate(final long size) {
    HeapMemory.instance().executeInArea(new Runnable() {
      public void run() {
        Encapsulate.this.target = new VTMemory(size);
      }
    });
  }

  public void encapsulate(final Class logicClass) {
    if (Encapsulate.implementsRunnable(logicClass)) {
      HeapMemory.instance().executeInArea(new Runnable () {
        public void run()
          /*
             Runs in heap, and enters a scope
          */
          Runnable step3 = new Step3(logicClass, target);
          while (true) {
            try {
              target.joinAndEnter(step3);
              break;
            } catch (InterruptedException ie) {
                continue;  // try again.
            }
          }
        }
      });
    }


    else {
      throw new IllegalArgumentException(
        "The logicClass must implement Runnable");
    }
  }

  public void waitForCompletion() throws InterruptedException {
    target.join();
  }

  /**
   * Make sure the logic clsas is a real class that implements
   * Runnable.
   */

  static private boolean implementsRunnable(Class logicClass) {
    Class [] interfaces = logicClass.getInterfaces();
    Class runnableInterface = Runnable.class;
```

```
    for (int i = 0; i < interfaces.length; ++i) {
      if (interfaces[i].equals(runnableInterface)) {
        return true;
      }
    }
    return false;
  }
}
```

Ejemplo 9. La clase de cola del hilo encapsulado

Es interesante rastrear el área de memoria que el patrón de hilo encapsulado crea con cada objeto.

Objeto	Área de memoria
La instancia de Encapsulate	Contexto de asignación actual de la función padre
El Runnable anónimo creado en el constructor de Encapsulate	Contexto de asignación actual de la función padre
El target **Scoped Memory**	Heap
Instancia de Step3	Heap
rtt	Target

Figura 4. Áreas de asignación para el ejemplo 9

2.2.5.- Garbage Collector

El entorno Java no proporcionará información de tipo dinámico o estático para caracterizar el comportamiento y las dependencias de cualquier algoritmo de recolección de basura proporcionado por el sistema. Sin embargo, *RTSJ* aún respetando esta premisa, sugiere proporcionar cierta información mediante métodos o subclases de *GarbageCollector.* Todos estos métodos sugeridos deberán estar soportados en todas las implementaciones.

Constructor

public **GarbageCollector**()

Métodos

public abstract RelativeTime **getPreemptionLatency**()

Instancias de *RealtimeThread* pueden expulsar de la *CPU* la ejecución del colector de basura (instancias de *NoHeapRealtimeThread* expulsan inmediatamente pero instancias de *RealtimeThread* deben esperar hasta que el colector de basura alcanza un punto de expulsión seguro). La latencia de expulsión es una medida del tiempo máximo que un *RealtimeThread* debe esperar hasta que el colector de basura alcanza un punto de expulsión seguro.

2.2.6.- Memorias restringidas anidadas

La máquina virtual de java necesita mantener en todo momento una estructura para saber qué objetos están ubicados dentro de ella; manteniendo también la naturaleza de cada uno de ellos y el orden en que han sido introducidos.

Para llevar este orden se pone en uno una estructura de pila, con crecimiento ascendente y que mantendrá en todo momento un puntero al área de memoria que está en uso en cada momento. (Figura 5)

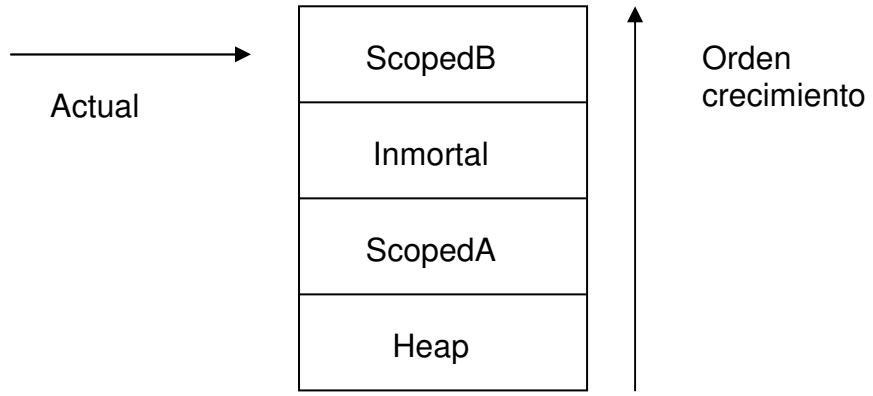


Figura 5. Pila

El puntero que representa el área de memoria que actualmente está en uso puede modificarse, la forma de hacerlo es con *executeInArea* llamado desde el área de memoria que va a coger el uso de la pila; si es de tipo *Scoped* cambiará la posición del puntero como ocurre en la figura *execute*; mientras que si el parámetro desde que se ha llamado a *executeInArea* es de tipo *Heap* o Inmortal, se creará una nueva pila de Memorias, con el área de memoria referenciado en la base(Figura 6).

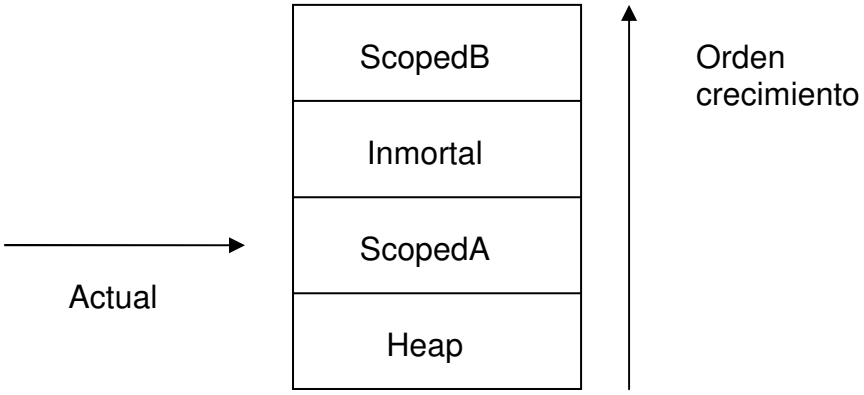


Figura 6. execute

Cada vez que un objeto entra en el área de memoria se introduce en al pila con su identificador y su naturaleza. El método para realizar esto es ***enter(),*** ejecutado en el área de memoria que va a introducirse en la pila, cambiando al estructura de la pila, creándose si el área tratado como actual no está en le cima, una estructura similar a un cactus (Figura 7).

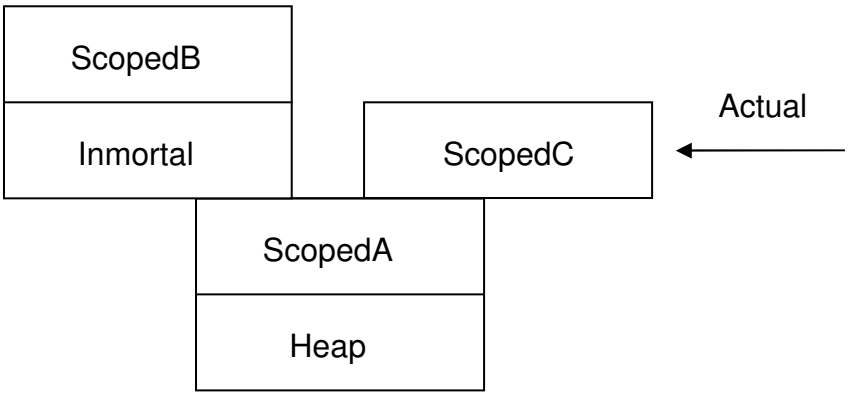


Figura 7. enter

Cada acción realizada en la pila de áreas de memoria están sujetas a la reglas de uso de la pila, como puede ser la regla del único padre, impidiendo posibles excepciones de tipo *ScopedCycleException.*

Compartición de regiones anidadas

Gracias a la naturaleza de la estructura del cactus varios hilos de ejecución pueden estar activos en el mismo momento, teniendo entre ellos áreas de memoria compartidas, respetando en todo momento las reglas de asignación básicas recogidas en la tabla de asignación. En la figura 8 se ve como dos hilos en ejecución comparten áreas de memoria.

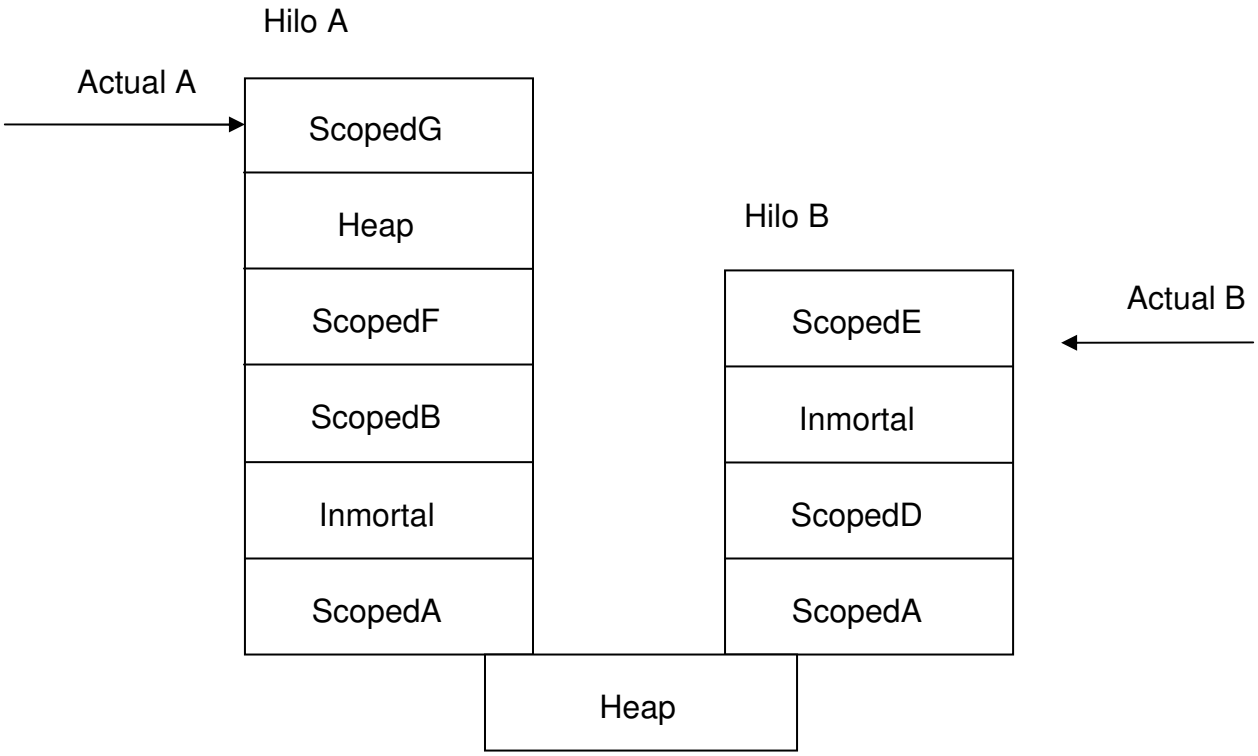


Figura 8. Compartición de áreas de memoria por dos hilos

Como se ve en la figura 8 existen dos hilos en ejecución a la vez, poseyendo cada uno una pila propia.

Debido a que en todo momento se gestionan las acciones bajo las reglas básicas de la gestión de memoria plantemos una posible situación de riesgo para la estructura de los hilos en ejecución. Suponemos que el hilo A intentase ejecutar el área de memoria de tipo *Scoped* E, situado en el hilo B y con padre dentro de ese hilo al área de memoria de tipo *Scoped* D. Se produciría una violación del área del único padre si se intentará directamente a partir del área de memoria Actual en A, ya que se pondría como padre de E en el hilo A al área de memoria de tipo *Scoped* D.

Para evitar esta situación de violación de la regla del único padre será necesario que dentro del hilo de ejecución A se cambie el puntero actual al *Scoped* A mediante *executedInArea.ScopedA*, produciéndose la situación de la figura 9, en que ya estará preparado en hilo de ejecución2 de generar una estructura válida para poder entrar el área de memoria de tipo *Scoped* E.

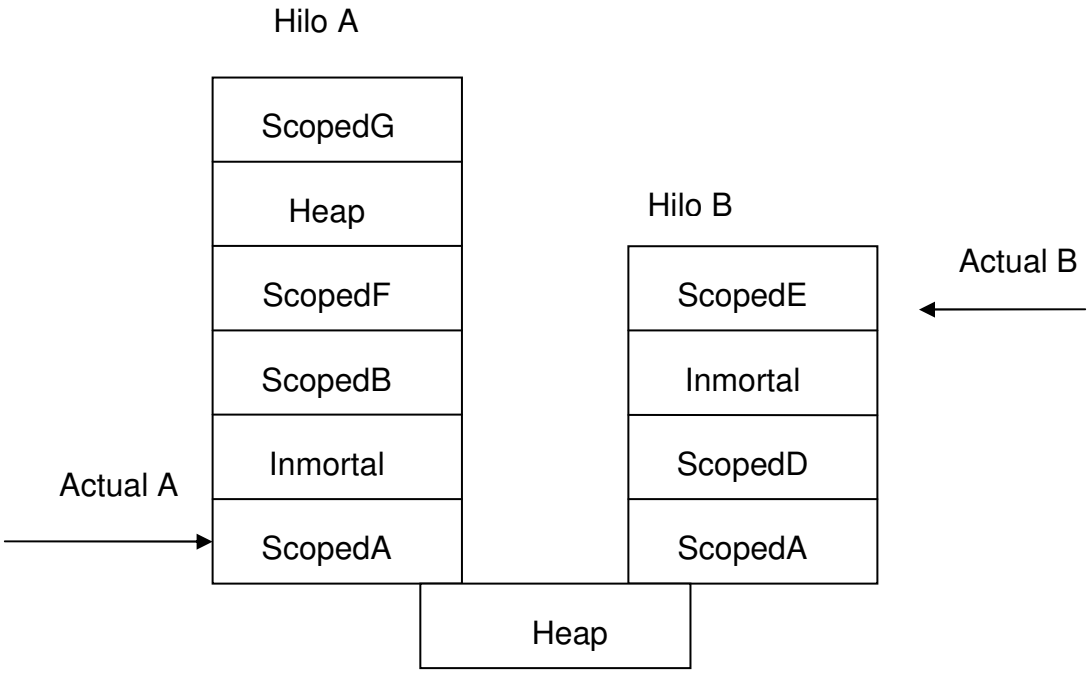


Figura 9. Generación de estructura válida

Una vez en esta situación se procede a la entrada del área *Scoped* D, esto es necesario para mantener la regla del único padre, ya que como en el hilo B el *Scoped* E generado previamente posee como padre al *Scoped* D, será necesario replicar esa situación dentro del hilo de ejecución A.

Para ello *Scoped* D llamará dentro del hilo A al método ***enter(),*** y tras esto se podrá llamar de la misma forma desde el *Scoped* E al mismo método quedando una estructura con los punteros actuales de A y B como se muestra en la figura 10.

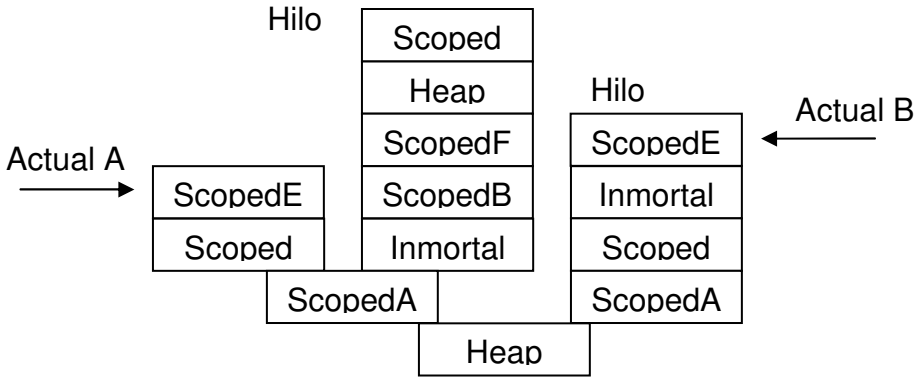


Figura 10. Actualización punteros

Herencia de la pila

La pila siempre se crea a la vez que el primer objeto en la maquina virtual, el valor de la pila puede ser modificado en su inicio cambiando los parámetros del área que entra en el hilo de ejecución inicial. Dependerá por tanto de la naturaleza del área de memoria inicial y del área actual en el contexto del hilo.

Si el área en uso en el momento de crear la pila es el *Heap*, si el área de memoria que creo al pila es de tipo *Heap* la pila nueva contendrá en su cima únicamente el mismo área de memoria desde el que se instancia, es decir, el *Heap*. Por otro lado bajo las mismas condiciones del área de memoria actual, si el área de memoria que se ejecuta no fuese de tipo *Heap*, la pila contendrá tanto el *Heap* como el área de memoria desde que se ha lanzado.

Otra posibilidad es que el área de memoria en uso en el instante de crear la nueva pila sea la memoria inmortal, en cuyo caso sucedería algo similar al caso de arriba, si es la misma memoria inmortal la que genera la nueva pila, en ésta solo contendrá esta memoria inmortal; mientras que si fuese de otro tipo contendría tanto a la memoria inmortal como a la memoria inicial en llamar a la nueva pila.

Casos diferentes son los sucedidos en caso de que el área de memoria en uso en el momento en que se ejecute la llamada sea de tipo *Scoped*, donde podrán suceder dos cosas:

- Si el área desde el que se llama es el mismo que está actualmente en uso o null, la nueva pila será la misma que la que posea el padre del área de memoria, que será de tipo *Scoped*, sumando el área de memoria actualmente en uso.

- Si el área desde el que se llama es distinto al que está actualmente en uso, la nueva pila será la del padre del área de memoria que está actualmente en uso, que será de tipo Scoped, más el área de memoria desde el que se ha llamado a la función.

3.- Simulación del sistema RTSJ

Scoped Memory permite mayor control en la liberación de objetos sin el colector de basura haciéndolo en tiempo de ejecución. El colector de basura empeora en exceso el tiempo de ejecución de las aplicaciones empeorando el rendimiento en tiempo y espacio.

Tras un largo proceso de aprendizaje en el entorno de la máquina virtual de java, comenzamos a realizar una implementación basada en un artículo de *Angelo Corsaro y Ron K. Cytron* [1].

3.1.- Modelo inicial

Objetivo: Creación de regiones por un hilo e interacción entre ellas

El primer objetivo que nos planteamos es el entendimiento del comportamiento de las propuestas que utiliza el paradigma *Scoped Memory*. En base a esto obtendremos los conocimientos necesarios sobre la naturaleza de las distintas regiones de memoria y las posibles características existentes entre ellas. Basado en los capítulos 1 y 5 de [5].

El segundo objetivo de esta implementación es lograr una buena creación e interacción entre las distintas regiones creadas por un hilo, pudiendo ser éstas de los tipos *Heap*, *Inmortal* y *Scoped*. Estos primeros pasos de uso de las regiones de memoria servirán para lograr una buena base abstracta de conocimiento de las distintas posibilidades, a la hora de implementar las distintas regiones de memoria.

Para llevar a cabo este primer modelo contaremos con un único hilo, el del programa principal, que se encargará de generar un número determinado de *Memory Areas*, pudiendo ser éstas de cualquiera de los 3 tipos. Esta creación será de forma aleatoria, teniendo la misma probabilidad de generarse los diferentes tipos de áreas de memoria.

Restricciones: El número de regiones de memoria que hemos elegidos es de diez, pudiéndose modificar a elección del usuario de la aplicación, pero hemos elegido este número ya que facilita la comprensión del modelo y a la vez proporciona la capacidad necesaria para abordar los distintas acciones a ver.

Una implementación *RTSJ* fuerza a la validación de una *Scoped* antes de ejecutar una asignación. Ésta validación consistirá en comprobar que un *Memory Area* con un tiempo de vida largo no pueda crear referencias a un objeto alojado en otro *Memory Area* de tiempo de vida menor. De ahí que *RTSJ* establezca una relación de parentesco entre *Scoped Memory* áreas, que es llamada *Regla del Único Padre*.

Se comprueba esta regla cada vez que un *Scoped Memory* área intenta hacer un *enter()*, y se explora la pila de *Scoped* en cada intento de crear una referencia. Es importante implementar las comprobaciones de las reglas de asignación de forma eficiente y predecible, ya que las referencias a objetos ocurren frecuentemente.

Por cada creación de las áreas de memoria asignamos a cada región un padre, respetando la Regla del Único Padre. Cada vez que un área es entrado por un hilo se ejecuta el método *checksingleParentRule(MemoryArea ma, ArrayList scopeStack)* que dotará a cada región de un único padre, esté método se explica con más detalle posteriormente. A su vez se asignarán tres referencias a otras *Memory Areas* del árbol en ese momento existente también de forma

aleatoria; el fin de estas referencias será la comprobación de la regla de las asignaciones ilegales entre las distintas áreas de memoria.

El principal problema de estas implementaciones era que no podíamos tener un programa o una serie de programas que requiriesen memoria de forma indeterminada y concurrente. La forma de solventar estas características, fundamentales a la hora de probar eficientemente nuestros modelos, era la inserción en cada acción límite de la aleatoriedad.

Tanto la creación como la asignación a regiones de memoria para las referencias está hecho de forma aleatoria, ya que pensamos que es una buena forma de aproximarnos al modelo de memoria contando con un único hilo que no hace demanda de regiones de memoria para la ejecución de sus tareas. La introducción de la aleatoriedad nos permite comprobar la validez de nuestra implementación, ya que en cada caso obtenemos un resultado distinto y válido, es decir, todas las simulaciones son independientes e indeterminadas, no nos limitamos a probar con un número determinado de casos prueba.

Este factor de aleatoriedad se podría quitar fácilmente si el usuario de la aplicación lo desea forzando situaciones requeridas por este, lo cual ha sido necesario para poder corregir los errores de la implementación. La forma sería, por ejemplo, dando toda la probabilidad de crearse áreas *Scoped*, por lo que nunca se generarán regiones de tipo *Inmortal* o *Heap.*

A continuación pasamos a detallar como hemos llevado a cabo el desarrollo para esta implementación comentando cada una de las clases con sus atributos y la relación que guardan entre ellas.

3.1.1.- Memory Area

Es la clase abstracta de la que heredan *Inmortal Memory*, la *Heap Memory* y los áreas de la *Scoped Memory*.

Para poder llevar a cabo la implementación del modelo los objetos de la clase Memory Area necesitarán tener los siguientes atributos.

Atributos:

private MemoryArea parent;

Necesitamos este atributo de tipo *MemoryArea* denominado *parent* para referirnos al padre de cada región creada.

private int depth;

Poseerá también un atributo de naturaleza entera *depth* que indicará la profundidad del área *Scoped* dentro del árbol de referencias.

private String id;

Hemos capacitado a esta clase con un identificador de objeto, de tipo *String* que nos ayudará a identificar cada *MemoryArea*.

private MemoryArea[] referencias;

El atributo MemoryArea[] referencias nos ayuda a simular las asignaciones entre *Memory Areas*, en principio hemos puesto un número fijo de referencias a cada *Memory Area*, 3. El número de referencias se refiere a referencias entre regiones, es una abstracción del modelo *RTSJ* para estudiar el comportamiento del mismo.

Constructor

public **MemoryArea**()

Generamos un nuevo objeto de tipo *MemoryArea*. Inicializamos la profundidad de cada *MemoryArea* a –1 ya que asumimos que tanto *Heap* como *Inmortal* tienen profundidad -1, el padre a null, y las referencias a null.

Métodos

Hemos definidos tanto los métodos accesores como los mutadores de cada atributo de un *MemoryArea*.

3.1.2.- Heap Memory

Extiende de la clase *MemoryArea*, heredando todos sus atributos.

3.1.3.- Inmortal Memory

Extiende de la clase *MemoryArea*, heredando todos sus atributos.

3.1.4.- Scope Memory

Extiende de la clase *MemoryArea*, heredando todos sus atributos.

En relación a las *ScopeMemory*, hemos de garantizar una propiedad básica; la regla del único padre (*Single Parent Rule*), garantizando que cada *Scoped Memory* tiene al menos un padre y sólo será ese durante toda su vida. Para garantizar esta propiedad comprobamos que se cumpla esta regla cada vez que se entra en una *Scoped.*

Por otro lado mediante el algoritmo de chequear las referencias válidas (*CheckReferenceValidity*) comprobará la validez de las asignaciones desde un determinado área de memoria a objetos alojados en otras áreas distintas.

Los métodos *SingleParentRule* y *CheckReferenceValidity* los comentaremos de forma más detallada cuando expliquemos la clase *Main*.

Atributos

private int refcount;

El atributo *refcount* de naturaleza entera llevará la cuenta de los hilos que referencian esa región. Se incrementará cuando el hilo entre en la región.

En este caso el contador de referencias valdrá como mucho uno ya que sólo contamos con un hilo de ejecución para esta implementación.

Cuando el contador pasa de 1 a 0 todos los objetos nuevos serán recolectados. En esta primera simulación no se ha implementado la recolección.

private ScopeMemory parentScoped;

Es un puntero al padre de tipo *Scoped Memory* de este objeto. Asignamos este puntero con la generación de un árbol cuyos nodos son sólo regiones de tipo *Scoped.*

private ArrayList display=new ArrayList() ;

Guardamos en un array los antecesores en el árbol; este display sólo contendrá regiones de tipo *Scoped Memory*.

Constructor

public **ScopeMemory**()

Crearemos un objeto de tipo *Scoped Memory* inicializando *refcount* a 0, ya que inicialmente no hay ninguna referencia a esta región, *parentScoped* a null y el display de regiones de memoria de tipo Scoped vacío.

Métodos

Accesores y mutadores de los atributos de la clase.

3.1.5.- Main

Clase principal del modelo cuya principal función será la de simular el único hilo de ejecución encargado de realizar las acciones. Esta clase llevará a cabo el mantenimiento de dos estructuras de datos de tipo árbol, uno en el que se guardarán todos los Memory Areas con sus relaciones de parentesco, mientras que el otro será un subárbol creado a partir del primero en el que sólo se mostrarán las áreas de memoria de tipo Scoped.

El ciclo de simulación de este modelo va a seguir un esquema como éste (Figura 11):

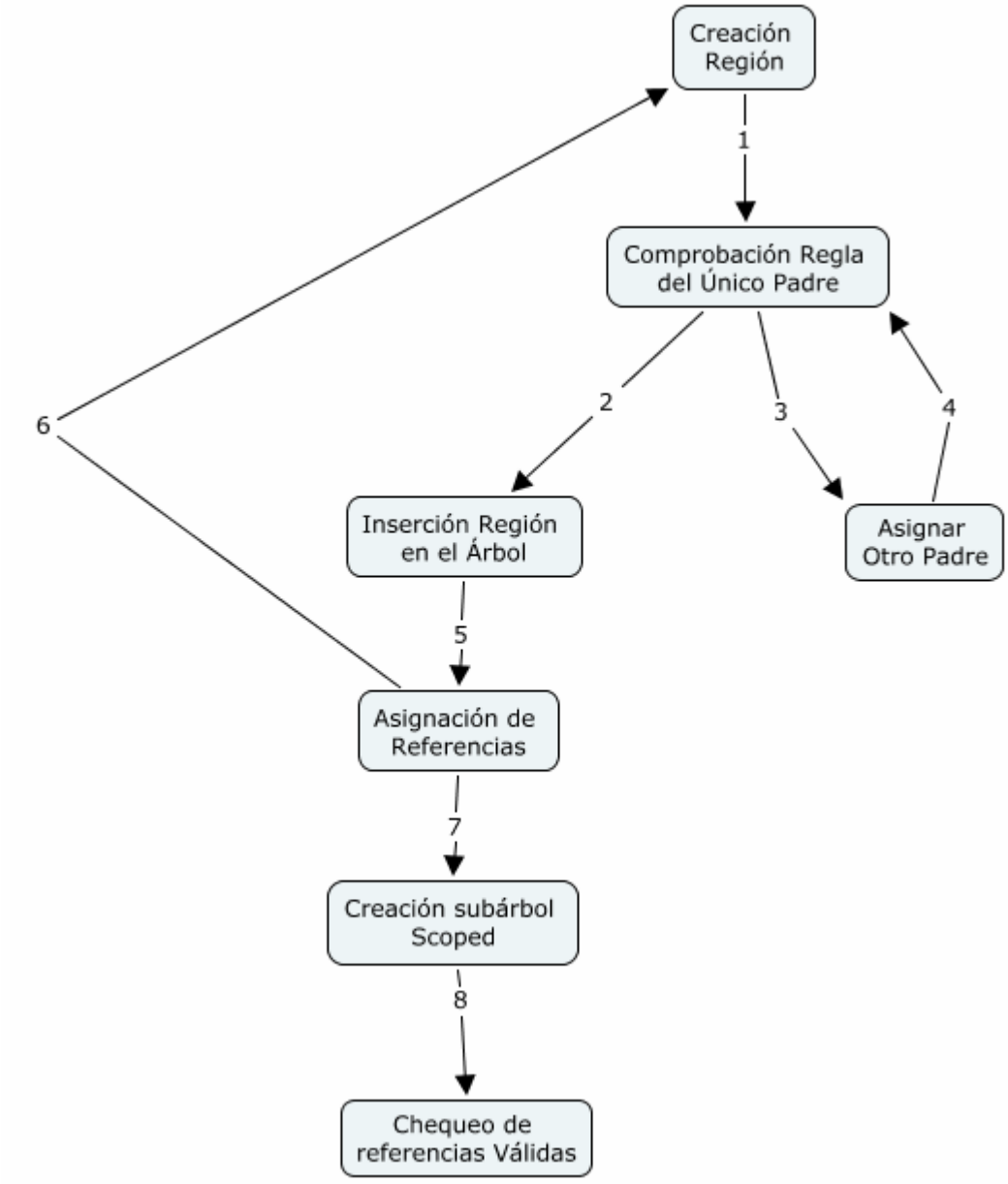


Figura 11. Ciclo de simulación

El hilo de ejecución *main* creará una región de un determinado tipo asignado de forma aleatoria (*Scoped Memory*, *Inmortal* o *Heap*) teniendo todos los tipos la misma probabilidad de crearse. Esta parte de la implementación simula el método *MemoryArea.enter()* de los hilos.

A continuación se comprueba la Regla del Único Padre (1). Los parámetros de esta regla serán la región que se acaba de crear y un array que contiene todas las regiones creadas anteriormente. Si el resultado de la regla es *true,* insertamos la nueva región en el árbol de *Memory Areas* (2). En caso de que la comprobación de la regla del único padre fuese fallida (3) se procedería a volver a intentar buscar un padre válido (4). El siguiente paso tras la inserción válida será la de asignar a esta área de memoria tres referencias a otras áreas de memoria de forma aleatoria (5). Este ciclo se realizará x veces a elección del usuario, en nuestro caso hemos elegido 10 iteraciones (6).

Una vez creado el árbol completo de *Memory Areas*, que refleja como ha entrado el hilo en las distintas regiones a lo largo del proceso de ejecución, se crea un subárbol en donde solo estarán contenidas las regiones *Scoped* en el que mostramos el *display* de cada una (sus antecesores) y su profundidad (7).

En base a las referencias aleatorias asignadas en el momento en el que una región es entrada comprobaremos la validez de éstas sacando por pantalla el resultado (8).

Métodos

checkSingleParentRule

Para simular la regla del único padre hemos necesitado implementar los dos métodos siguientes:

public static ScopeMemory ***findFirstScope****(ArrayList scopeStack)*

Devuelve el primer objeto de tipo *ScopeMemory* existente en la *scopeStack* pasada por parámetro. En nuestro caso la estructura pasada por parámetro es un array que contiene todas las regiones creadas hasta el momento. Si no hubiese ningún área de memoria de tipo *ScopeMemory* el resultado sería la devolución del Primordial Scoped o *Heap.*

```
public static ScopeMemory findFirstScope(ArrayList scopeStack) {
      ScopeMemory firstScope;
      firstScope = PrimordialScope;
      for (int i = scopeStack.size(); i > 0; i--) {
            if (scopeStack.get(i - 1) instanceof ScopeMemory) {
                  ScopeMemory sm = (ScopeMemory) scopeStack.get(i - 1);
                  firstScope = sm;
                  break;
            }
      }
      return firstScope;}
```

Algoritmo 4 .findFirstScoped

En el siguiente ejemplo se muestra una posible pila de regiones para ver el funcionamiento del método y por tanto la región de área *Scoped* devuelta al ejecutarse. En a) tenemos el *Heap* y dos regiones de tipo *Scoped*, se devolverá por tanto la región situada más cerca de la cima de la pila, es decir *Scoped* A. En b) tenemos una pila en la que no existe ningún objeto de tipo *Scoped*, se devolverá el área de memoria Primordial (Figura 12).

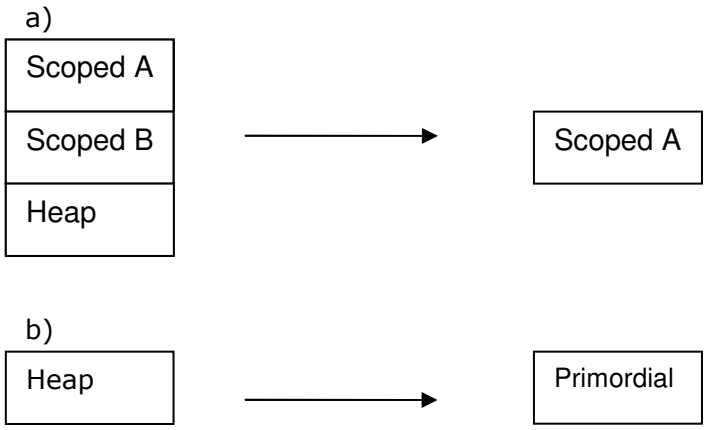


Figura 12. Pila de regiones

public static boolean ***checkSingleParentRule****(MemoryArea ma, ArrayList scopeStack)*

Este método va a tener como parámetros el *Memory Area* creado *ma*, y un array que contendrá todas las regiones existentes en ese momento. Lo primero es comprobar que el área de memoria *ma* es de tipo *Scoped Memory*. Mediante el método *findFirstScoped*, se recoge el último *Scoped Memory* entrado por el hilo. Si el padre de *ma* es null o es el obtenido mediante el método *findFirstScoped* asignaremos a *ma* ese padre y se añadirá al array *ScopedStack* ma, incrementando su contador de referencias en uno y devolviendo como resultado true; en otro caso devolverá false. En caso de que el área de memoria no sea de tipo *Scoped Memory* se escogerá como padre una de las regiones entradas anteriormente por el hilo.

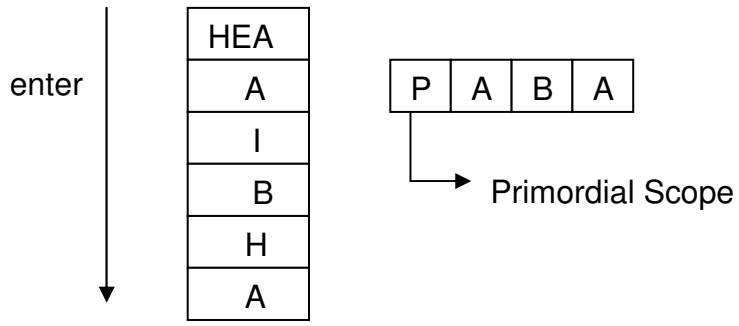


Figura 13: Ejemplo de violación de la regla del único padre

La figura muestra la existencia de dos áreas de memoria de tipo Scoped entrada por un hilo T1. Se viola la regla ya que ABA sería la relación de parentesco: pretende que B sea padre de A, pero A ya tiene padre, el *Heap (PrimordialScope)* (Figura 13).

```
public static boolean checkSingleParentRule(MemoryArea ma,ArrayList
scopeStack) {

      boolean isSingleParentRuleOK = true;
      MemoryArea parent;
      if (ma instanceof ScopeMemory) {
         ScopeMemory sm = (ScopeMemory) ma;
         parent = findFirstScope(scopeStack);
         if ((sm.getParent() == null) || (sm.getParent() == parent)) {
            sm.setParent(parent);
            scopeStack.add(sm);
            sm.setRefcount(sm.getRefcount() + 1);
          } else
            isSingleParentRuleOK = false;
      }

      return isSingleParentRuleOK;
}
```

Algoritmo 5. CheckSingleParentRule

MemoryReferenceChecks

Para mantener la seguridad de Java y permitir referencias pendientes existen una serie de reglas restrictivas de aplicación sobre el área del *ScopedMemory*:

- Una referencia a un objeto en la *ScopedMemory* nunca puede ser almacenado en un objeto alojado en el Java *Heap.*

- Una referencia a un objeto en la *ScopedMemory* nunca puede ser almacenado en un objeto alojado en la *InmortalMemory.*

- RTSJ nos da un algoritmo para validar las referencias a memoria, una aproximación está descrita en el Algoritmo 6, comprobando que el área de memoria donde está creado el hilo del *Scoped* fue cogido después del área de memoria desde el que se llamó.

- Las referencias a objetos de la memoria *Inmortal* o del *Heap* deben ser almacenados dentro de un objeto alojado en el área de la *Scoped Memory.*

Para simular la validez de las referencias hemos implementado el siguiente método:

> *public static boolean* ***checkReferenceValidity****(MemoryArea from, MemoryArea to)*

Como parámetros de este método existirán dos regiones de memoria, FROM que es el *Scoped Memory* del que parte la referencia y TO que será el área de memoria que aloja el objeto al que se está referenciado.

El método devolverá true en el caso en el que el FROM esté por debajo de TO dentro del árbol de regiones o en el caso en que la profundidad de TO sea igual a -1, es decir que la región que contiene el objeto referenciado sea de tipo *Inmortal* o *Heap (Primordial Scope)*. En cualquier otro caso devolverá false.

Por ejemplo suponiendo que todos los nodos representados en la Figura 14 Son de tipo *Scoped*, y que a una profundidad -1 se encontrarán las áreas de memoria de tipo *Inmortal* y *Heap*, sólo podrán existir referencias de tal forma que el nodo origen esté en los niveles de profundidad inferiores con respecto del nodo destino. De tal forma que aparecen en rojo las posibles referencias inválidas dentro de este posible árbol de *Scoped*, apareciendo por otro lado en verde las referencias permitidas, ya que siempre irán de abajo arriba éstas últimas.

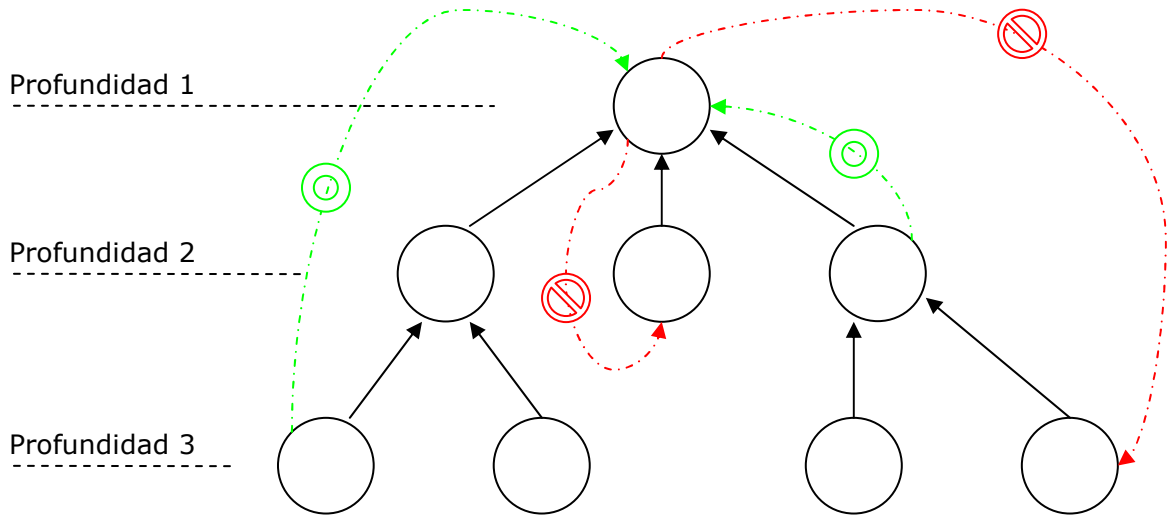


Figura 14. Ejemplo referencias válidas

El algoritmo que comprobará estas referencias será:

```
public static boolean checkReferenceValidity(MemoryArea from,
MemoryArea to) {

      boolean validReference = false;
      if ((from.getDepth()) >= (to.getDepth())) {
            if (to.getDepth() == -1)
                  validReference = true;
            else {
                  ScopeMemory smf = (ScopeMemory) from;
                  ScopeMemory smt = (ScopeMemory) to;
                  String idf =
                  smf.getDisplay().get(to.getDepth()).toString();
                  String idt = smt.getId();
                  if (idf.compareTo(idt) == 0)
                        validReference = true;
                  }
            }
            return validReference;
}
```

Algoritmo 6. CheckReferenceValidity

Nueva comprobación de Referencias Válidas

Cualquier referencia de un Scope podrá ser definida como el camino desde la raíz hasta su ubicación en el interior del árbol, estas referencias en el camino son indistintamente de memoria inmortal y de las del *Heap*. *Parenthoodtree* va a ser una relación con el ancestro del árbol incluyendo información en los hijos como puede ser la profundidad, de la misma forma sólo contendrá información de los nodos que pertenezcan a áreas de memoria de la *Scoped.* De esta forma se evita tener un mapa de memoria completa, mejorando la eficiencia de los algoritmos; de la misma forma se consigue en tiempo constante una comprobación de las referencias de las áreas de memoria.

Otro de los métodos que hemos implementado en el *main* es el que mostramos a continuación, que nos permite obtener el subárbol formado sólo por regiones de tipo *Scoped Memory* a partir del árbol de parentesco inicial en el que aparecen regiones de los tres tipos. No es una transformación del árbol inicial, si no que seleccionamos solamente los *Memory Area* que nos interesa mostrar, los de tipo *Scoped*. Mantendremos la estructura de los dos árboles durante todo el tiempo de ejecución del sistema.

public static void ***construyeSubarbol****(ArrayList graph)*

Método que a partir del árbol general de dependencias y orden de entrada en las *MemoryArea* instanciadas todas ellas en el ArrayList *graph* pasado como parámetro, genera el subgrafo compuesto por los áreas de memoria de tipo *Scoped Memory* estableciendo sus dependencias en función del grafo de origen.

Para ello, para cada *Scoped* vamos a guardar un *display* en el que contendrá las *Scoped* que estén por encima suya y estén conectadas, es decir, que guarde el orden de generación, y la profundidad en que se encuentran en el subárbol.

Por último el método hará una impresión en la pantalla estructurando los pasos anteriores; devolviendo cada *display* y profundidad de cada *Scoped Memory* y por tanto la estructura del subárbol. La forma de identificar cada nodo será mediante su identificador único para cada uno, que será de tipo *String*.

3.1.6.- Diagrama de interacción entre clases

En el siguiente diagrama de clases UML mostramos la interacción entre las distintas clases implementadas para esta primera versión.

Como hemos comentado anteriormente tenemos una clase abstracta *MemoryArea* de la que heredan *ScopeMemory, Heap* e *InmortalMemory*.
En las clases *MemoryArea* y *ScopeMemory* mostramos los atributos, explicados en puntos anteriores. Omitimos por simplificar el diseño los métodos de ambas clases, ya que son los accesores y mutadores de éstos.
Por último tenemos la clase *Main* con los métodos que contiene, las flechas discontinuas hacia las clases *ScopeMemory*, *Heap* e *InmortalMemory* significa que hace uso de éstas (Figura 15).

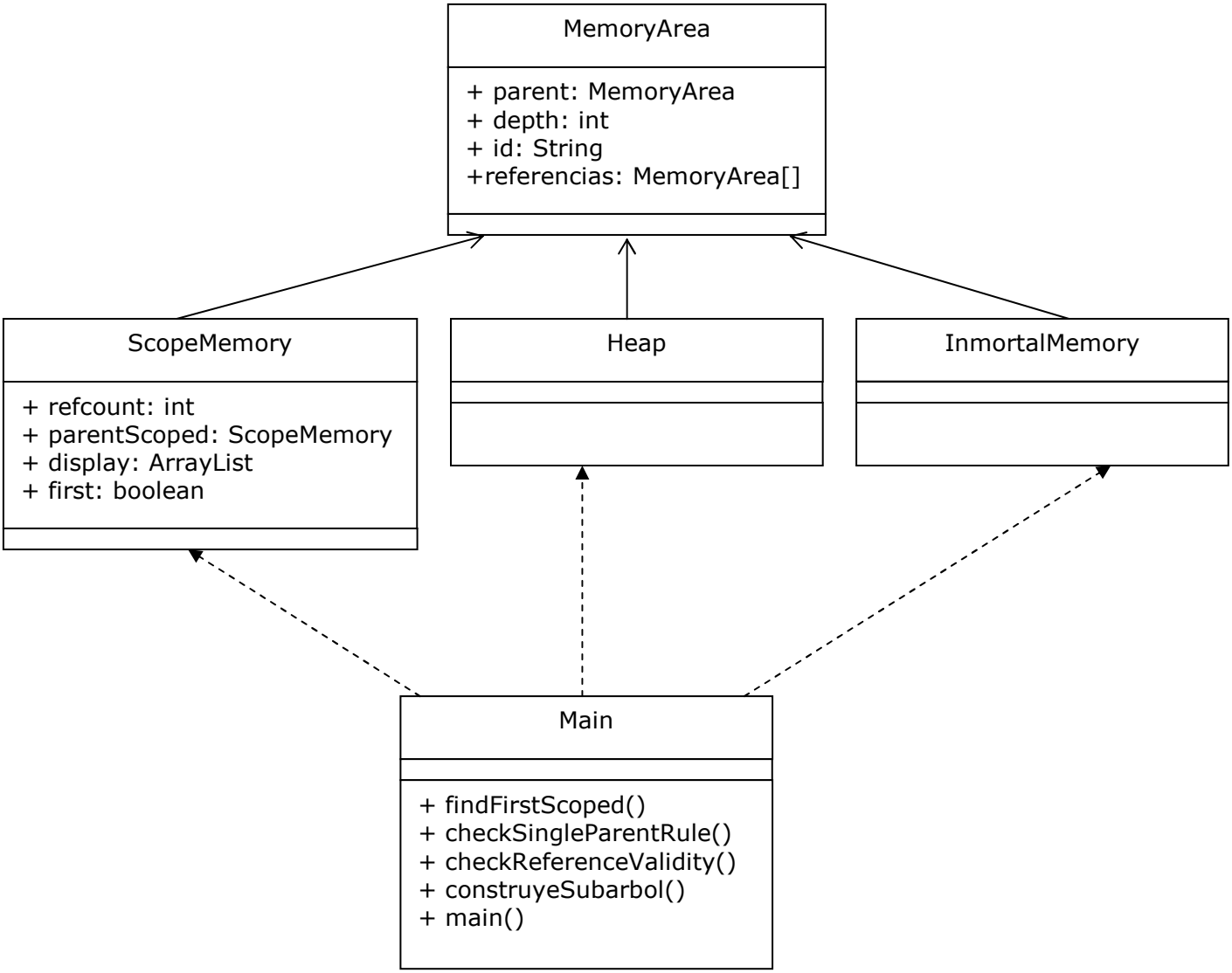


Figura 15. Diagrama clases

3.1.7.- Ejemplo Aplicación

En el presente ejemplo vamos a ilustrar las diferentes acciones realizadas durante un ciclo de ejecución y la salida final tras los métodos comentados anteriormente.

Tras ejecutar la aplicación en una plataforma “eclipse” nos ha dado como un resultado posible la siguiente estructura de árbol de *Memory Areas*(*Figura 8*); como elección por parte del usuario hemos elegido que se creen diez regiones de memoria y que a la hora de tipificar éstas tengan la misma probabilidad de

ser Scoped, Inmortal y Heap, así como que el número de referencias realizada por cada Memory Area sea de tres.

En esta figura están representadas las *Memory Area* con la abreviación de su tipo junto con un número que identifica en el orden en que se han sido entradas por parte del hilo.

De este modo *SMX* identificará un área de memoria de tipo *Scoped Memory* creado en el momento *X*.

El *Texto 1* refleja el orden en el que hilo ha entrado en las distintas regiones identificado con un número por cada región.

```
El hilo entra en SM0
El hilo entra en SM1
El hilo entra en HEAP
El hilo entra en SM3
El hilo entra en HEAP
El hilo entra en SM5
El hilo entra en SM6
El hilo entra en HEAP
El hilo entra en HEAP
El hilo entra en SM9
El hilo entra en SM10
```

Texto 1. Árbol parentesco

Cada vez que una región de memoria es entrada por el hilo comprobamos la regla del único padre. El método que implementa esta regla asigna un padre a cada región, que será único durante todo el tiempo de vida de ésta. A partir de la relación de parentesco así creada obtenemos el árbol de dependencias mostrado en la Figura 16.

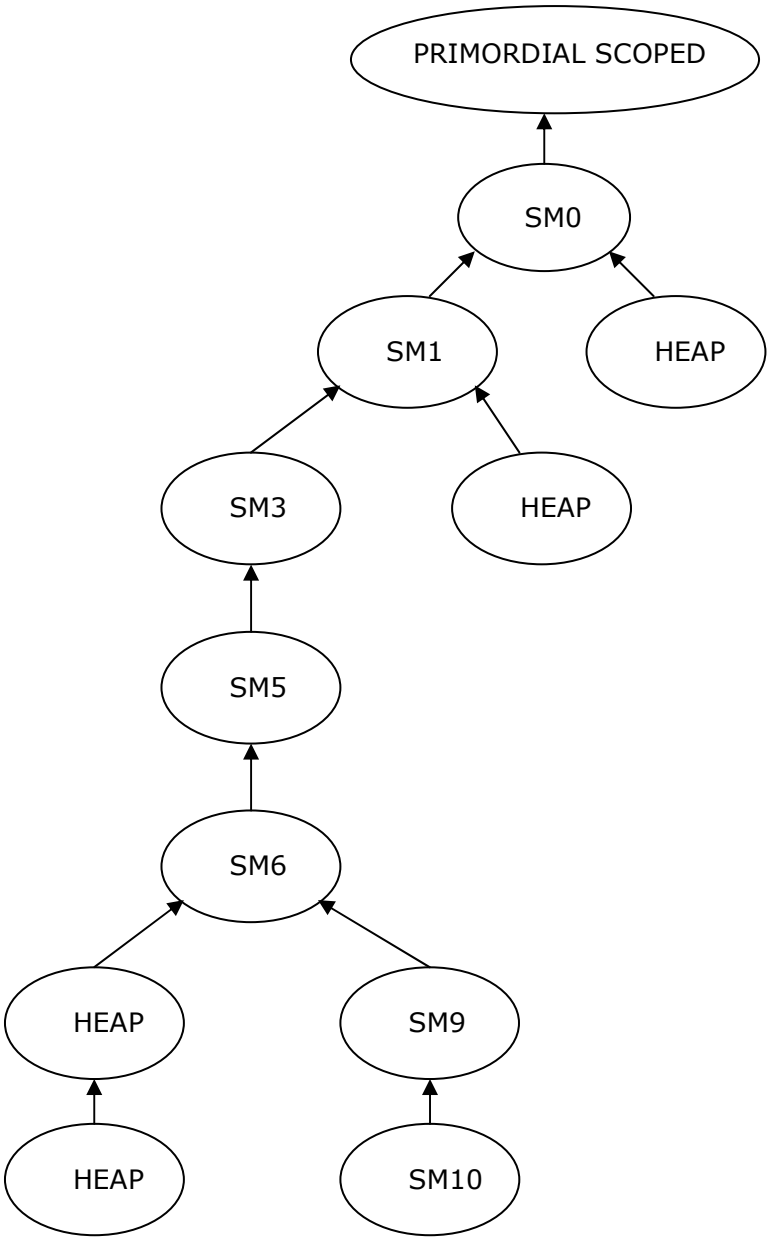


Figura 16: Árbol dependencias

El hilo entra en SM0 como no ha entrado en ninguna región anteriormente la Regla del Único Padre le asigna como padre el *PrimordialScope*.
A continuación el hilo entra en SM1, la Regla del Único Padre le asigna como padre el último Scoped entrado, es decir, SM0.
El hilo entra en HEAP, al no ser una región de tipo Scoped se le asigna como padre una de las regiones entradas anteriormente.
El hilo entra en SM3, la Regla del Único Padre le asigna como padre el último Scoped entrado, es decir, SM1.
El hilo entra en HEAP, al no ser una región de tipo Scoped se le asigna como padre una de las regiones entradas anteriormente.

El hilo entra en SM5, la Regla del Único Padre le asigna como padre el último Scoped entrado, es decir, SM3.
El hilo entra en SM6, la Regla del Único Padre le asigna como padre el último Scoped entrado, es decir, SM5.
El hilo entra en HEAP, al no ser una región de tipo Scoped se le asigna como padre una de las regiones entradas anteriormente.
El hilo entra en HEAP, al no ser una región de tipo Scoped se le asigna como padre una de las regiones entradas anteriormente.
El hilo entra en SM9, la Regla del Único Padre le asigna como padre el último Scoped entrado, es decir, SM6.
El hilo entra en SM10, la Regla del Único Padre le asigna como padre el último Scoped entrado, es decir, SM9.

Una vez generado el árbol el siguiente paso es proceder a calcular el subárbol dejando de lado los áreas de memoria que no nos interesan, es decir, los tipos Heap e *Inmortal,* fijándonos solamente en aquellas regiones de tipo *Scoped*.

El árbol resultante (*Figura 17*) muestra la relación de parentesco entre las regiones de memoria de tipo Scoped. Se asignará a cada nodo o región una profundidad dentro de este subárbol y un display que contendrá la identidad de todos sus antecesores.

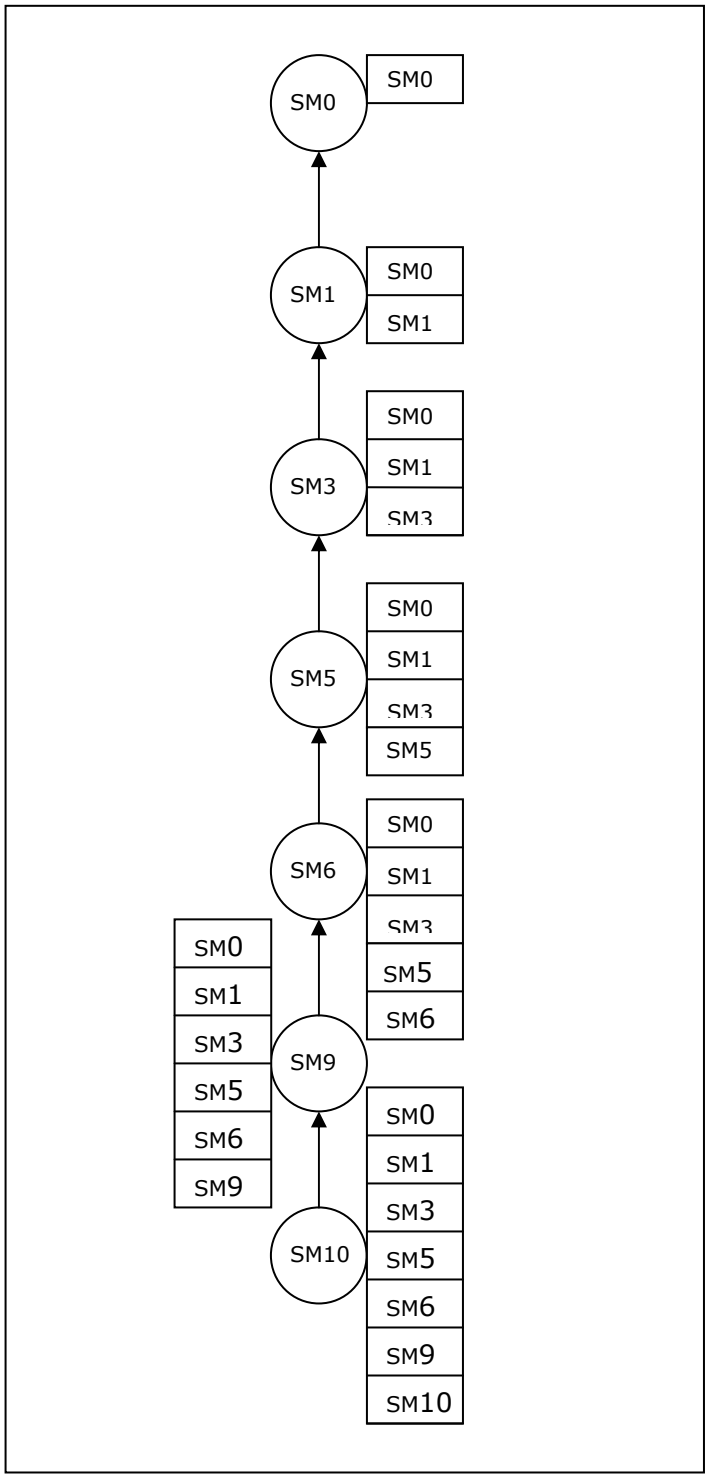


Figura 17. Árbol Scope

Los *displays* almacenarán el conjunto de áreas de memoria con las que tienen parentesco, y que tendrán que recorrer para destruirse él mismo.
La forma de recorrido en este caso sería de abajo a arriba de la estructura del *display*.

Se puede observar en la consola los datos obtenidos tras la generación de este subárbol, siendo los datos generados:

```
El nodo SM0 tiene como scope padre a : PrimordialScope
Display : [SM0]
Profundidad : 0

El nodo SM1 tiene como scope padre a : SM0
Display : [SM0, SM1]
Profundidad : 1

El nodo SM3 tiene como scope padre a : SM1
Display : [SM0, SM1, SM3]
Profundidad : 2

El nodo SM5 tiene como scope padre a : SM3
Display : [SM0, SM1, SM3, SM5]
Profundidad : 3

El nodo SM6 tiene como scope padre a : SM5
Display : [SM0, SM1, SM3, SM5, SM6]
Profundidad : 4

El nodo SM9 tiene como scope padre a : SM6
Display : [SM0, SM1, SM3, SM5, SM6, SM9]
Profundidad : 5

El nodo SM10 tiene como scope padre a : SM9
Display : [SM0, SM1, SM3, SM5, SM6, SM9, SM10]
Profundidad : 6
```

Texto 2. Subárbol Scope

Se puede observar que junto con el *display* previamente comentado aparece para cada *Scoped Memory* la profundidad a la que se sitúa dentro del subárbol, suponiendo que la profundidad donde se encuentra la raíz, *PrimordialScoped*, es 0.

Por último en la aplicación comprobamos si las referencias son válidas.

```
Chequear referencias válidas del nodo : SM0 al nodo HEAP
true
Chequear referencias válidas del nodo : SM0 al nodo SM9
false
Chequear referencias válidas del nodo : SM0 al nodo HEAP
true
```

H2, SM9 y H7 serán las regiones de memoria, elegidas de forma aleatoria, que contendrán los objetos a los que tienen referencias objetos alojados en SM0.
Referencias de un objeto alojado en un Scoped a un objeto alojado en el Heap o en una región Inmortal están permitidas.
Referencias de un objeto alojado en un Scoped a otro objeto alojado en un Scoped situado por encima en el árbol de parentesco están permitidas.

Referencias de un objeto alojado en un Scoped a otro objeto alojado en un Scoped no situado por encima en el árbol de parentesco no están permitidas.
A continuación mostramos el resto de resultados obtenidos por consola de nuestro árbol de parentesco de ejemplo.

```
Chequear referencias válidas del nodo : SM1 al nodo HEAP
true
Chequear referencias válidas del nodo : SM1 al nodo SM1
true
Chequear referencias válidas del nodo : SM1 al nodo SM5
false
Chequear referencias válidas del nodo : HEAP al nodo HEAP
true
Chequear referencias válidas del nodo : HEAP al nodo SM9
false
Chequear referencias válidas del nodo : HEAP al nodo SM9
false
Chequear referencias válidas del nodo : SM3 al nodo SM9
false
Chequear referencias válidas del nodo : SM3 al nodo HEAP
true
Chequear referencias válidas del nodo : SM3 al nodo SM6
false
Chequear referencias válidas del nodo : HEAP al nodo SM1
false
Chequear referencias válidas del nodo : HEAP al nodo HEAP
true
Chequear referencias válidas del nodo : HEAP al nodo SM0
false
Chequear referencias válidas del nodo : SM5 al nodo SM5
true
Chequear referencias válidas del nodo : SM5 al nodo SM0
true
Chequear referencias válidas del nodo : SM5 al nodo SM1
true
Chequear referencias válidas del nodo : SM6 al nodo HEAP
true
Chequear referencias válidas del nodo : SM6 al nodo SM1
true
Chequear referencias válidas del nodo : SM6 al nodo SM6
true
Chequear referencias válidas del nodo : HEAP al nodo SM6
false
Chequear referencias válidas del nodo : HEAP al nodo SM5
false
Chequear referencias válidas del nodo : HEAP al nodo HEAP
true
Chequear referencias válidas del nodo : HEAP al nodo HEAP
true
Chequear referencias válidas del nodo : HEAP al nodo HEAP
true
Chequear referencias válidas del nodo : HEAP al nodo SM1
false
Chequear referencias válidas del nodo : SM9 al nodo HEAP
true
Chequear referencias válidas del nodo : SM9 al nodo HEAP
true
Chequear referencias válidas del nodo : SM9 al nodo HEAP
true
```

```
Chequear referencias válidas del nodo : SM10 al nodo HEAP
true
Chequear referencias válidas del nodo : SM10 al nodo HEAP
true
Chequear referencias válidas del nodo : SM10 al nodo SM1
true
```

Texto 3. Chequeo referencias

Esto completaría el ciclo de simulación de esta primera implementación. Nos ha servido como primer acercamiento al modelo de memoria RTJS y entendimiento de éste. Conseguimos simular la creación de regiones por un único hilo de ejecución, el del programa principal, y la relación de parentesco establecida entre estas regiones cada vez que el hilo iba entrando en ellas y cada vez que una requería de un objeto alojado en otra.
Para hacer una simulación más próxima a la realidad nos planteamos mejorar esta primera implementación ampliando su funcionalidad. Aumentando el número de hilos que podrían crear y entrar en distintas regiones e implementando la funcionalidad del recolector de basura. Estas mejoras han dado lugar a la implementación descrita en el siguiente punto 3.2.

3.2.- Solución al modelo

Creación de regiones por varios hilos e interacción entre ellos

3.2.1.- Objetivo de la implementación

Nuestro objetivo es crear un modelo sobre la distribución de memoria, de la que harán uso los hilos. El número de hilos que interactuarán al mismo tiempo en ejecución lo elegirá el usuario. *Heap*, *Inmortal*, y *Scoped* van a ser, como ha ocurrido en el modelo anterior, los distintos tipos de memoria que vamos a utilizar. Crearemos hilos, que serán los encargados de crear las regiones de memoria o de acceder a ellas, aumentando las referencias de las regiones *Scoped*. Para visualizarlo cómodamente sacaremos por pantalla un grafo de las distribuciones.

El motivo que nos ha impulsado a introducir hilos de ejecución en la implementación ha sido el intento de simular más fielmente el modelo de programación utilizando las regiones de RTSJ. De este modo mientras que en la JVM están continuamente pidiendo y cediendo memoria una serie de programas sin que la máquina virtual pueda predecir cada petición o acción referente a la memoria, con la inserción de los hilos y la naturaleza aleatoria de

las acciones de éstos con respecto a la naturaleza de la memoria vamos a poder simular de una forma más cercana un entorno real.

3.2.2.- Filosofía del modelo

En la implementación de este modelo vamos a hacer uso de las regiones de memoria y de los hilos. Cuando se inicia el modelo hay solamente una única región de memoria, el *HEAP*, y los hilos se irán creando a medida que pase el tiempo a elección del usuario mediante la interfaz proporcionada en la implementación. Cuando un hilo se crea podrá hacer dos cosas mientras viva: crear nuevas regiones de memoria o acceder a regiones ya existentes. Hay un momento en que el hilo termina todo lo que tiene que hacer y, entonces, muere, este conjunto de acciones podrá ser fácilmente modificado por parte del usuario, en nuestro caso hemos decidido dotar al hilo de 10 acciones que, como hemos comentado, se realizarán bajo una naturaleza aleatoria.

Tenemos tres tipos de regiones: *Heap*, *Inmortal* y *Scoped*. De estas tres la que más nos interesa es la región *Scoped*, y es sobre la que basaremos todo el modelo. Cuando un hilo X se crea puede, o bien crear regiones o bien acceder a ellas, siendo las tres regiones igualmente importantes, pero solo será la región *Scoped* sobre la que tendremos funciones independientes: mirar la regla del único padre, aumentar las referencias, modificar los hilos que acceden al *Scoped*, etc., como se muestra en el ciclo de ejecución comentado en la sección del modelo anterior, en la figura 11, ya que cada hilo en sus 10 acciones predefinidas, realizará el conjunto de acciones ilustradas en esta figura.

Al crear el primer hilo, la única región que tiene disponible es el *HEAP*. Si referencia solo podrá referenciar al *HEAP*, y si crea una región nueva el *HEAP* será su padre. Pero en el momento de crear una nueva región está solicitando memoria para crear otra región distinta al *HEAP* (habrá dos regiones: *HEAP* y la nueva), y sobre ésta podrá tanto crear como referenciar. El resto de los hilos que se creen funcionarán de la misma manera que el primero.

Todas las regiones de memoria disponen de un puntero al padre, para facilitar la implementación.

Otra clave para el éxito y la ventaja de la utilización de múltiples hilos de ejecución en una aplicación, o aplicación *multithreaded*, es que pueden comunicarse entre sí. Se pueden diseñar hilos para utilizar objetos comunes, que cada hilo puede manipular independientemente de los otros hilos de ejecución.

En nuestro caso los hilos se ejecutarán de forma concurrente de tal forma que accederán todos a la estructura de array que representa, en una primera instancia, al árbol de Memory Area. Para evitar un mal uso de los hilos y sobretodo de la región compartida es necesario sincronizar el conjunto de hilos que estén en ejecución. Con esta implementación el número de hilos será

dinámico durante todo el tiempo que quiera el usuario; ya que se podrán crear tanto como el usuario requiera tanto mientras otros hilos están creando MemoryAreas, como cuando está parada toda función de entradas o referenciaciones de hilos, como tras una recolección de uno o un conjunto de hilos.

La manera para que los hilos se ejecutan en paralelo y no interfieran entre ellos es mediante la creación de una región crítica a la que solo pueda acceder un hilo cada vez. Como solo puede acceder uno, el resto de hilos permanecen en un estado *wait* esperando a que termine el hilo que está dentro. Es en esta región donde los hilos crean o referencian; una vez que un hilo ha terminado de realizar todas sus acciones realizará una transmisión al resto de hilos de esa sección crítica informando de que la deja y por lo tanto otro hilo podrá entrar en ella, esta transmisión se realiza mediante un *notify.*

Mediante el uso de la sección crítica en la ejecución de los hilos se impondrán reglas de acceso de tal forma que una situación como la de la figura 18 nunca podría darse. En esta figura si no existiese la sincronización de los hilos, dos hilos accederían a la vez en la sección critica, pudiendo insertar en el grafo una región de memoria, en este caso una Scoped Memory, y además siendo éstas la primera y segunda Area de Memoria en ser creada en el árbol respectivamente; esto producirá que se viole la regla del único padre, ya que como se observa SM2 va a tener dos padres distintos en esta estructura: el Heap y SM1

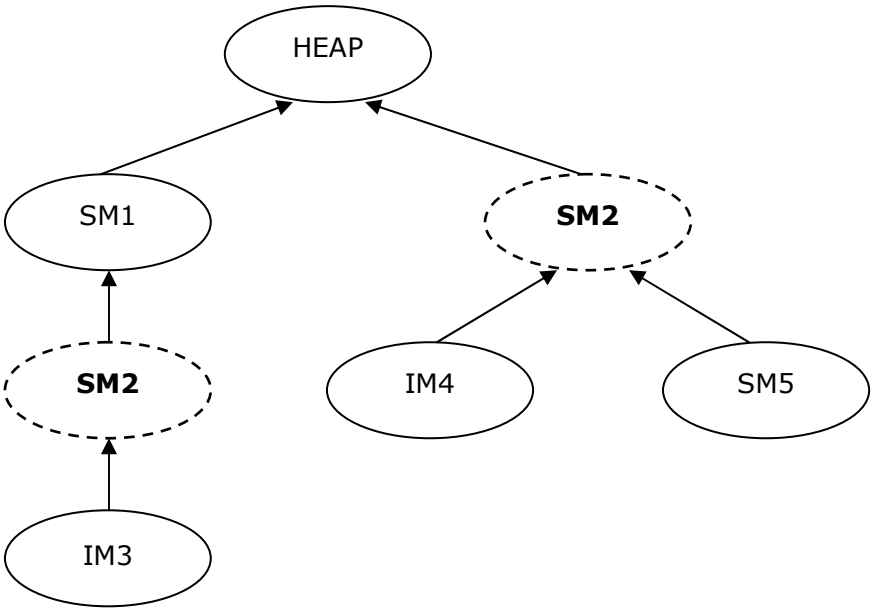


Figura 18. Violación regla del único padre mediante hilos.

Gracias a la inserción de la sincronización de los hilos y la nueva naturaleza del grafo como sección crítica, este problema, solucionado en el modelo anterior de tal forma que se garantizaba la existencia de un solo padre para cada Scoped Memory, será de nuevo solucionado pero esta vez para la ínter

actuación de varios hilos a la vez, evitando que se produzca la situación anterior y creando en vez de ese árbol el mostrado en la figura 19, en la que sí que se podrá producir que dos hilos distintos intentan hacer una entrada en la misma región de memoria Scoped, en este caso la 2, pero garantizará la regla del único padre y manteniendo su padre, Heap.

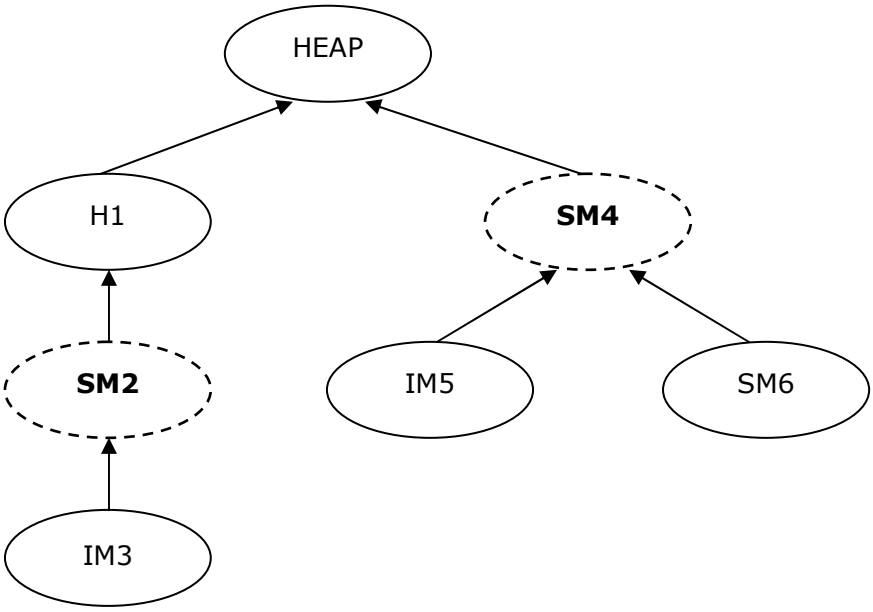


Figura19. Nombramiento de regiones distintas con el mismo nombre.

La forma en que hemos incluido los hilos de ejecución en la aplicación ha sido mediante el uso de monitores de Java; de tal forma que en nuestro caso como hemos comentado antes el grafo general va a ser la sección crítica a la que únicamente un hilo va a tener acceso en cada instante, manteniendo la estructura correcta del árbol de parentesco.

Cada hilo introducido será de una naturaleza como ésta:

```
ThreadImplementation nuevohilo =
new ThreadImplementation(Integer.toString(numerohilo),grafo,this);
```

De tal forma que los iremos enumerando con el fin de identificarlos a cada uno de una forma sencilla.

Ha tenido que ser necesario redefinir el método **run**(), predefinido para cada hilo, que será el método en cuyo cuerpo están las acciones que realizará cada hilo una vez tiene uso de la sección crítica, el grafo. Cada hilo tendrá predefinida diez acciones, que podrá ser cada una o crear una nueva Memory Area, o realizar una referencia entre dos regiones ya creadas. El pseudocódigo asociado a este método sería el siguiente.

```
public void run()

   para i=1 hasta 10//diez acciones predefinidas de cada hilo
    si  (crearMemoryArea)
                graph.meterMemoryArea(ma,true,this.getName())
    si(realizarreferencias)
                graph.meterMemoryArea(ma,false,this.getName())
   actualizainterfaz();
```

Como se puede ver en el código el método que posee la sincronización está asociado al grafo, ya que éste es el que desempeña la función de sección crítica.

El método más importante y el que va a tener la sincronización definida va a ser meterMemoryArea, perteneciente a la clase Grafo y que el pseudocódigo por el cual se va a gobernar la sincronización va a ser el siguiente. Comentamos únicamente lo necesario para lograr la exclusión mutua entre los hilos, mantenida mediante la variable booleana estaOcupado:

```
public synchronized int meterMemoryArea(MemoryArea nueva,boolean
crear,String nombre)

while( estaOcupado == true )
        {
        try {
            wait(); // Se sale cuando estaVacia cambia a false
        } catch( InterruptedException e ) {
            ;
            }
        }

Conjunto de acciones de inserción de Memory Area o creación de
referencias

estaOcupado = false;
    notify();
```

Este método va a ser por tanto aquel que se ejecute en la sección crítica, cediendo el grafo al resto de procesos ligeros una vez han terminado de realizar sus acciones, poniendo que ya no esta ocupada esta sección crítica, y comunicándoselo al resto de hilos mediante el *notify(), más abajo lo cometamos más a fondo.*

A las regiones *Scoped* les vamos a dar un trato especial.

1. *Contador de referencias*: Van a disponer de un contador de referencias que indica cuantos hilos contiene. Este contador es fundamental a la hora de reciclar las regiones *Scoped*, porque no podremos eliminar una región mientras su contador sea mayor que 0. Si el contador de la región

es mayor que 0 quiere decir que tiene dentro de ella procesos ligeros en ejecución. Si el contador de referencias está a 0 no tiene por qué eliminarse, puede tenerlo a 0 pero con hijos colgando de ella, y no puede reciclarse porque quedarían huérfanas, teniendo que esperarse su recolectado hasta que el conjunto de hijos colgantes de ella sean recolectados. Por lo que se ve, es muy importante tener un control estricto sobre el contador de referencias: incrementado o decrementado cuando sea necesario, y que solo pueda reciclarse en el momento adecuado.

2. *Regla del único padre*: Es muy importante implementar esta función en las regiones *Scoped* para controlar que los hilos no creen regiones fuera de su ámbito. Por ejemplo, si un hilo crea una región SM1 y más tarde crea otra región SM2, ésta tiene que ser hija de SM1; si luego crea otra región SM3 poniendo como padre a SM2, sólo podrá tener como a padre a SM2 hasta que muera, violando la regla del único padre en caso de que intentase tener como padre a HEAP o SM4. Figura 20.

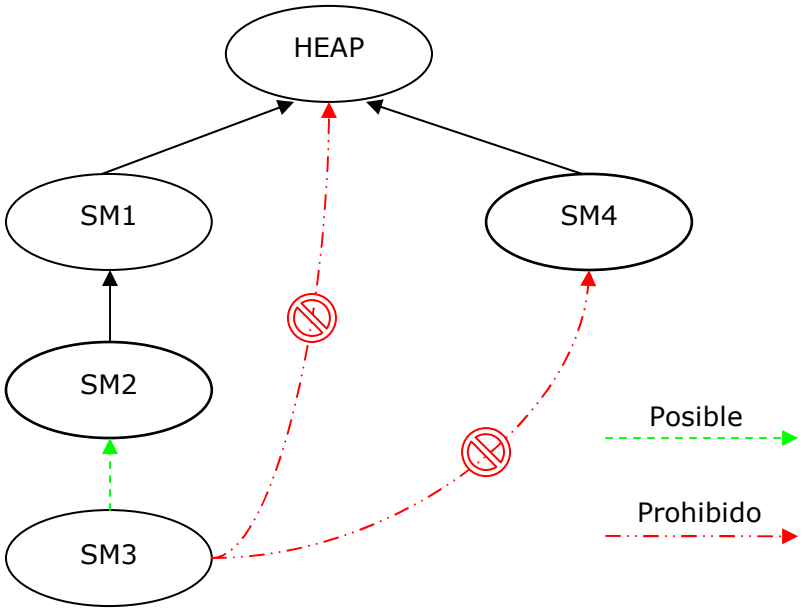
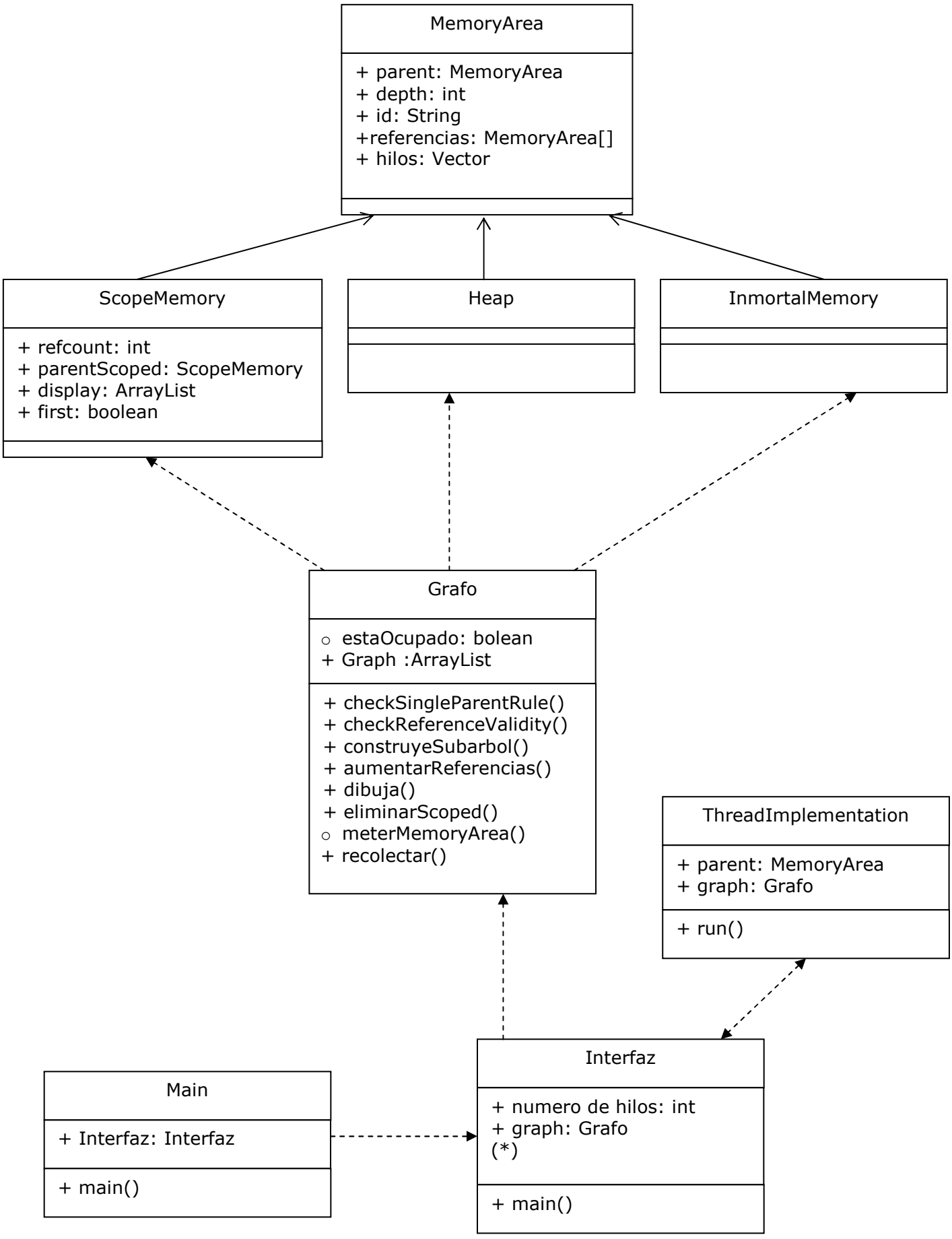


Figura 20. Ejemplo de referencias entre distintas regiones.

3.2.3.- Diagrama de interacción entre clases

En el siguiente diagrama de clases UML mostramos la interacción entre las distintas clases implementadas para esta primera versión.

Este diagrama va a ser una ampliación del de la sección 3.1.6. Hay que tener en cuenta que es una simplificación de las clases para no recargar el diseño, por ejemplo los métodos accesotes y mutadores no se muestran, indicándose otros métodos secundarios con (*), que van a ser los métodos y atributos necesarios para llevar a cabo la interfaz gráfica. Figura más abajo.


MemoryArea
+ parent: MemoryArea
+ depth: int
+ id: String
+referencias: MemoryArea[]
+ hilos: Vector
ScopeMemory
+ refcount: int
+ parentScoped: ScopeMemory
+ display: ArrayList
+ first: boolean
Heap
InmortalMemory
Grafo
○ estaOcupado: bolean
+ Graph :ArrayList
+ checkSingleParentRule()
+ checkReferenceValidity()
+ construyeSubarbol()
+ aumentarReferencias()
+ dibuja()
+ eliminarScoped()
○ meterMemoryArea()
+ recolectar()
ThreadImplementation
+ parent: MemoryArea
+ graph: Grafo
+ run()
Interfaz
+ numero de hilos: int
+ graph: Grafo
(*)
+ main()
Main
+ Interfaz: Interfaz
+ main()

3.2.4.- Método de simulación

Para generar este modelo vamos a simular las asignaciones de las regiones de memoria en los hilos. Estas simulaciones se basarán en una simulación ficticia lo más real posible, intentando acercarnos a un entorno real en que varios procedimientos están realizando acciones concurrentemente.

Basándonos en el modelo anterior vamos a realizar una serie de modificaciones en base a las características ya comentadas, insertando otras para poder crear el entorno de simulación en base a hilos. Vamos a comentar las características ya comentadas anteriormente e introduciendo las nuevas para este modelo.

Nos vamos a referir en cada caso a las nuevas clases que hemos tenido que introducir y al conjunto de clases modificadas en parte del modelo anterior.

3.2.4.1.- Regiones de Memoria

Los tres tipos de regiones de memoria de los que se va a hacer uso: *Heap*, *Inmortal* y *Scoped* heredarán de una región superior llamada *MemoryArea*. Ésta clase padre tendrá todos los atributos que tienen en común las tres regiones, como puede ser el nombre y el puntero al padre, y el puntero a la región más externa.

Las regiones *Heap* e *Inmortal* no tienen ninguna implementación específica porque solo sirven para hacer una simulación más real, los hilos pueden crear y referenciar también en estas regiones aunque no lo tendremos en cuenta.

La región *Scoped* sí que tiene una implementación propia, con atributos independientes del resto de regiones, como el contador de referencias antes comentado.

Un atributo nuevo de la clase Memory Area, que por tanto heredarán Scoped, Heap e Inmortal es, en este modelo el vector *hilos.* Este vector guarda todos los hilos que entran dentro de la región, inicialmente como se puede intuir estará vacío. La principal característica que se le atribuye a este atributo es que nos permitirá llevar un control más exhaustivo de la situación actual de cada Memory Area y por lo tanto del árbol de dependencias.

3.2.4.2.- Grafo

La principal función esta nueva clase es la de emular un contenedor para guardar todas las regiones de memoria que se van creando.

Al inicio de la implementación el grafo solo tendrá un nodo, el *HEAP*, y a medida que los hilos crean nuevas regiones se van insertando dentro. Se ha implementado como un ArrayList porque es la estructura de datos más

adecuada para esta implementación. El método de inserciones en el árbol es el mismo que en el modelo anterior, con la única diferencia que en este caso llevaremos el control de los diferentes hilos y la cuenta de los diferentes hilos que están en cada región de memoria.

La característica mas importante de esta clase es que se trata de la clase de cumple la sección crítica, es decir, el objeto de tipo grafo sólo podrá poseerlo y por tanto modificarlo como quiera un hilo en cada momento. Como hemos comentado anteriormente la variable de sincronización usada en esta clase es *estaOcupado*.Para realizar esta exclusión mutua hemos implementado estos hilos de ejecución mediante monitores.

El conjunto de atributos necesarios para esta clase serán:

- ArrayList graph: este va a ser el contenedor básico para este modelo comentado más arriba; irá almacenando el conjunto de Areas de Memorias creadas por cada hilo modificando a su vez las características de estas en manera de que sea necesario por las acciones realizadas en este grafo.

- **int** siguiente: este entero nos va a servir para ver en que orden son entrados el conjunto de Memory Areas en el grafo.

- **int** numeroScopes: se trata de un simple contador de Scoped.

- **int** area: variable auxiliar necesaria para realizar las acciones de esta clase.

- **boolean** estaOcupado = **false**: variable de sincronización usada por los hilos para evitar entrar dos en la sección crítica, inicialmente como se muestra esta puesta a falso, ya que en los primeros momentos ningún hilo posee la sección crítica. Cada vez que un hilo puede entrar en esta sección cambiará el valor de esta variable poniéndola a verdadero, hecho que evitará que el resto de hilos pudiesen acceder a este grafo Una vez un hilo ha terminado de realizar sus acciones la pondrá a falso estaOcupado, informando al resto de hilos mediante la función antes comentada *notify*

Los métodos más importantes creados en esta clase son:

*public synchronized int **meterMemoryArea**(MemoryArea nueva,boolean crear,String nombre)*

Los hilos se crearán a petición del usuario y las acciones que podrán realizar dentro de su sección crítica podrán ser: crear o referenciar regiones de memoria. Se ha creado un estándar para los hilos, de tal forma, que todos creen o referencien 10 veces (aunque se puede modificar). Para decidir si

crean o referencian tenemos una variable que usa el *random* y que concede la misma prioridad a las dos acciones, esta diferenciación será pasada por parámetro (crear). Otro dato pasado por parámetro va a ser el nombre de la región a realizar una acción.

- *Creación*: El hilo tiene otro random para decidir qué tipo de región de memoria va a crear: *Heap*, *Inmortal* o *Scoped*, siendo el *Scoped* el que tiene mayor prioridad, aunque casi imperceptible. Una vez decidido, el hilo entra en la región crítica y le asigna un nombre a la región nueva. Lo inserta en el grafo utilizando las funciones propias del ArrayList, le asigna el padre y aumenta las referencias en caso de que la nueva región sea *Scoped*, la naturaleza del Memory Area será otro dato pasado por parámetro para aligerar la sección critica de accions que pueden realizarse en el proceso de espera de cada hilo para que el grafo quede liberado.

- *Referenciación*: El hilo entra en la región crítica en la zona de referencias. En esta parte del código el hilo busca una región, usando la regla del único padre, a la que pueda referenciar. Si la región que va a referenciar es *Scoped* tiene que aumentar el contador de referencias de esa región si este hilo no la había referenciado antes.

Cuando el hilo ha terminado de hacer todas sus operaciones permanece latente mientras el usuario no diga lo contrario. Igual que en la creación, el hilo morirá a petición del usuario, recolectando sus Memory áreas como indicamos en una sección posterior.

public boolean ***checkSingleParentRule****(MemoryArea padre, String hilo)*

Este método comprobará de una manera similar que se cumpla la regla del único padre de forma similar a como lo hemos hecho en el modelo anterior, esta regla se fijará principalmente en dos cosas:

1. Que el hilo todavía no esté dentro del grafo, por lo que cualquier región que cree estará bien definida. En el caso de referenciar podrá hacerlo a cualquier región que haya dentro del grafo sin ninguna dificultad.

2. Si el hilo ya existe dentro del grafo, es decir, que ya ha creado o referenciado alguna región de memoria, tiene que comprobar que referencie o cree en la región adecuada. Un hilo sólo puede crear una región nueva si alguno de sus padres *Scoped* tienen el hilo dentro del vector *hilos*.

   Si el hilo ya existe en el grafo porque, o bien ha creado o bien ha referenciado a *heap* o *inmortal*, y ningún *Scoped* lo tiene, puede crear donde quiera.

   Pero si algún *Scoped* lo contiene en su vector *hilos* y el hilo quiere crear otra región *Scoped*, no podrá crearla en una rama

distinta de las que procedan de ese *Scoped*, porque cambiaría el padre.

La regla del único padre solamente comprueba las regiones *Scoped*, el resto de regiones pasan desapercibidas y solo sirven para hacer una simulación acorde con el modelo de programación RTSJ.

public void ***aumentarReferencias****(ScopeMemory ma,String nombre)*

Cada vez que un hilo entra en una región tiene que aumentar el contador de referencias de la propia región. Si el hilo crea una región el contador se pondrá a 1 y mostrará por pantalla el hilo que contiene, si referencia tiene que comprobar si la región ya contenía el hilo o no. En caso de que lo contuviera el contador se quedará igual, y si no lo incrementamos en 1. Para poder realizar este incremente en el número de referencias vamos a necesitar que se pasen por parámetro tanto el Scoped Memory para poder aumentarle el número de referencias como el nombre, que será la forma de identificar el área de memoria al que modificarle sus referencias.

Ponemos como ejemplo un árbol de dependencias entre regiones de memoria en que han sido necesario una serie de acciones por parte de cuatro hilos para poder llegar a la estructura de la figura 21.

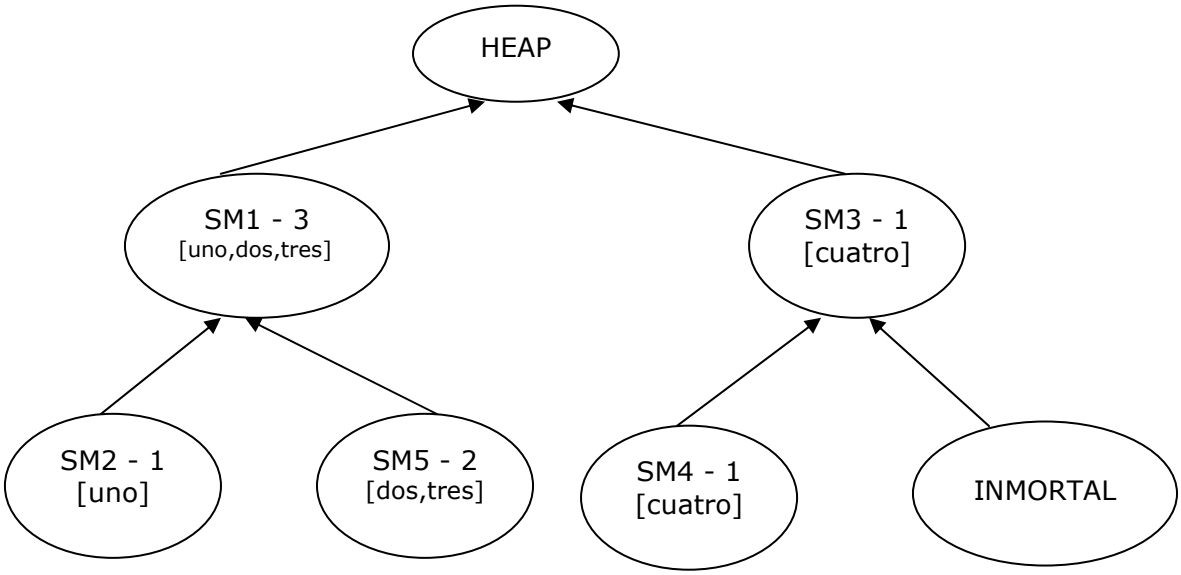


Figura 21. Ejemplo de cómo funciona la regla del único padre y el incremento del contador de referencias.

Explicación del ejemplo:

Cuando iniciamos el programa solo tenemos el *HEAP* como único nodo del grafo. El usuario introduce un nuevo hilo (*uno*) que iniciará su ronda de acciones (crear o referenciar). *Uno* crea la primera región *Scoped* SM1, metiendo en el vector *hilos* el nombre del hilo que lo crea e incrementando el contador de referencias en 1. El grafo quedaría de la siguiente manera (Figura 22):

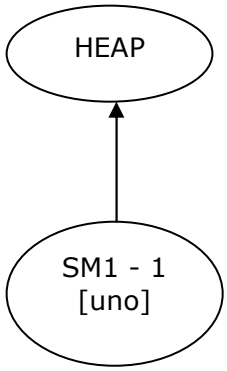


Figura 22. Introducción hilo 1

El hilo *uno* empezaría a crear referencias en el *HEAP* y en *SM1* hasta que vuelve a entrar otra región de memoria *Scoped* llamada *SM2*. Aumenta el contador de referencias de *SM2* porque es una región nueva que no contenía ningún hilo (Figura 23).

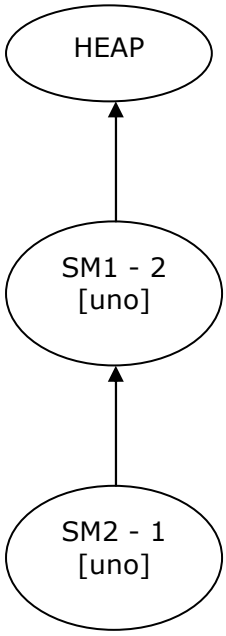


Figura 23. Inserción nueva Memory Area por hilo 1

Uno todavía no ha terminado su ciclo de acciones y sigue referenciando en *HEAP*, *SM1* o *SM2*. Finalmente termina y se queda latente hasta que el usuario no decida terminar con él.

El usuario decide meter tres hilos más, *dos*, *tres* y *cuatro*, que actuarán en paralelo. El primero de los tres en crear es *cuatro*, que genera la región *Scoped SM3* metiendo el hilo en su vector y aumentando el contador de referencias. El siguiente también es *cuatro* creando la región *Scoped SM4*, aumentando su contador (Figura 24).

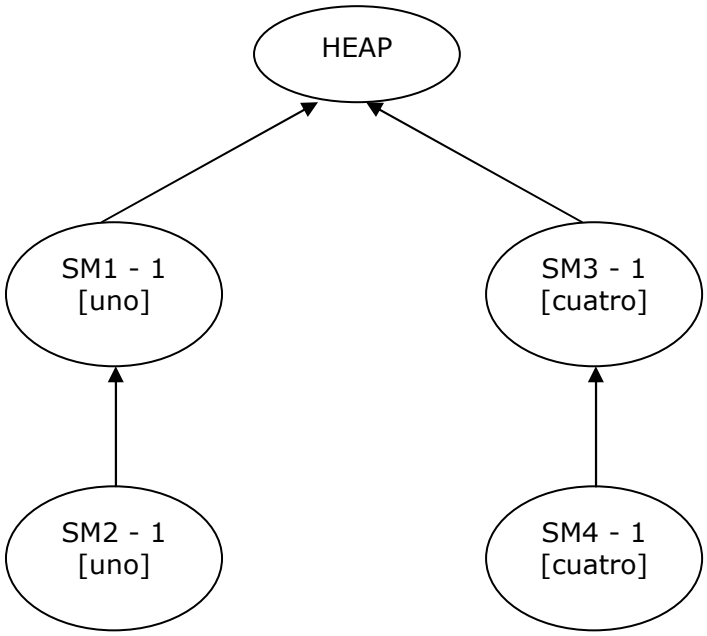


Figura 24. Introducción hilo 4

Le llega el turno al hilo *dos* de crear una región *Scoped SM5*, hija de *SM1*, con lo que incrementa su contador. Y el hilo *tres* crea una referencia sobre *SM1*, aumentando el contador de *SM1*.

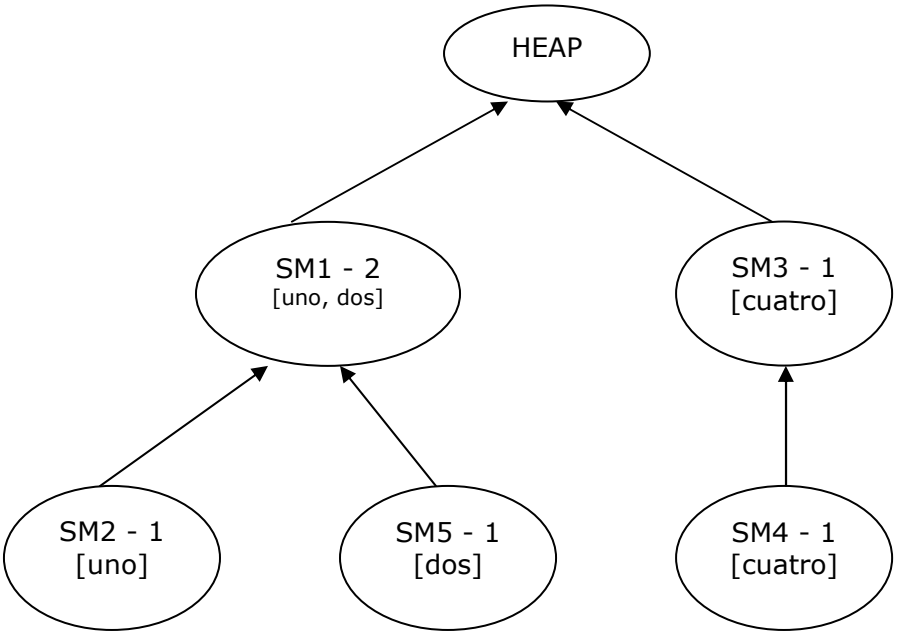


Figura 25. Introducción hilo 2

Finalmente el hilo *tres* referencia a *SM2*, y *cuatro* crea una región inmortal que es hija de *SM3*.

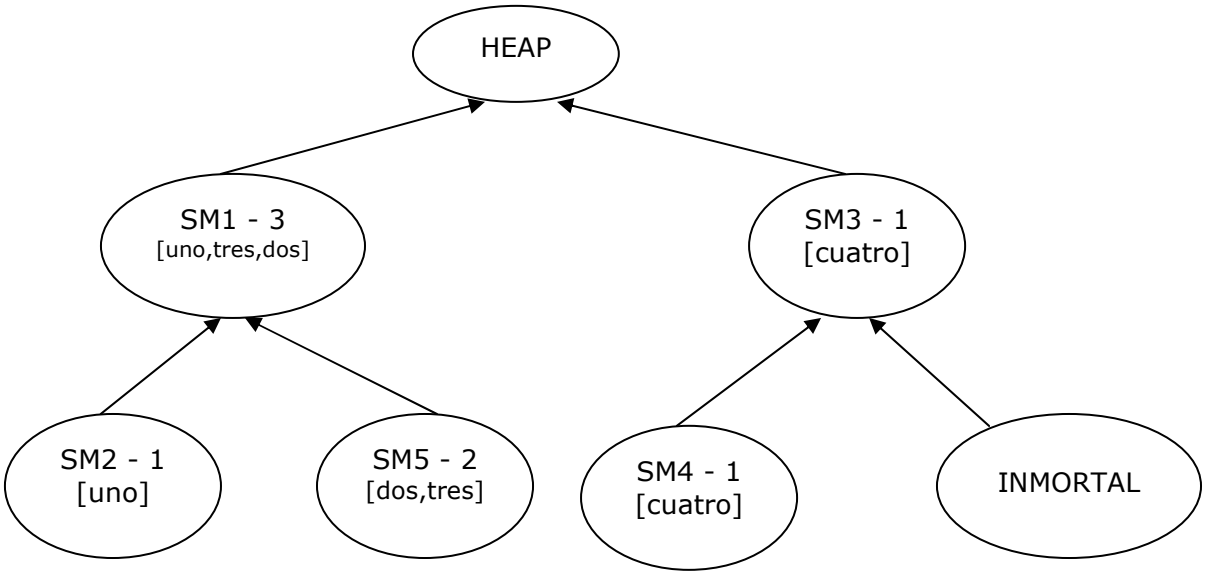


Figura 26. Introducción hilo 3

public void ***recolectar****(String hilo)*

Cada vez que un hilo desaparece de un *Scoped* su contador de referencias se decrementa en 1. Una región *Scoped* no puede reciclarse si tiene hijos *Scoped*. Por tanto, una región *Scoped* se recicla siempre y cuando no contenga hilos dentro y no cuelguen otras regiones de él.

Los hilos mueren a petición del usuario, él elige qué hilo quiere matar. El método que llevamos para eliminar un hilo del grafo es el siguiente:

1. Se pasa por parámetro el nombre del hilo que se quiere eliminar.

2. Se hace un recorrido de todo el grafo mirando qué regiones *Scoped* contienen el nombre del hilo en el vector de *hilos*.

3. Cuando se encuentra una región *Scoped* que contiene el hilo. Se decrementa en 1 su contador de referencias.

4. Si el contador de referencias del *Scoped* se queda en 0 y no tiene hijos se elimina la región mediante el método también definido en esta clase***:*** *public void* ***eliminarScope****(ScopeMemory sm,String hilo).*

Vamos a ilustrar un ejemplo de recolección en base al árbol generado en al apartado anterior:

El usuario decide matar el hilo *uno*. Se recorre el grafo para ver que nodos *Scoped* contienen este hilo y ejecutan el paso 3 anteriormente explicado. El grafo quedaría de la siguiente manera.

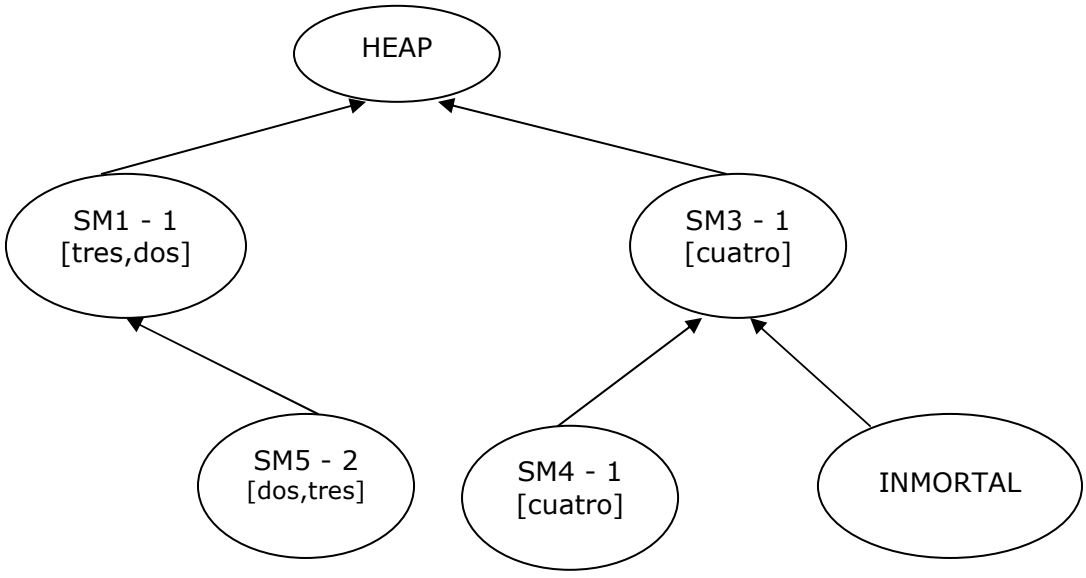


Figura 27. Recolección hilo 1

Como la región *SM2* solo contenía el hilo *uno* y no tenía hijos, desaparece.

Ahora el usuario decide eliminar el hilo *cuatro*, quedando:

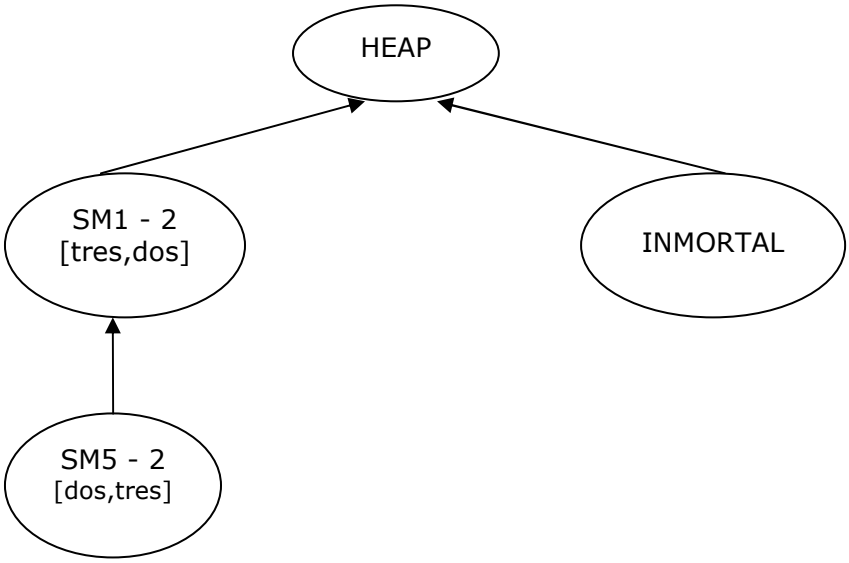


Figura 28. Recolección hilo 4

Se elimina el hilo *tres*, y como en *SM5* sigue estando el hilo *dos*, *SM1* no puede desaparecer.

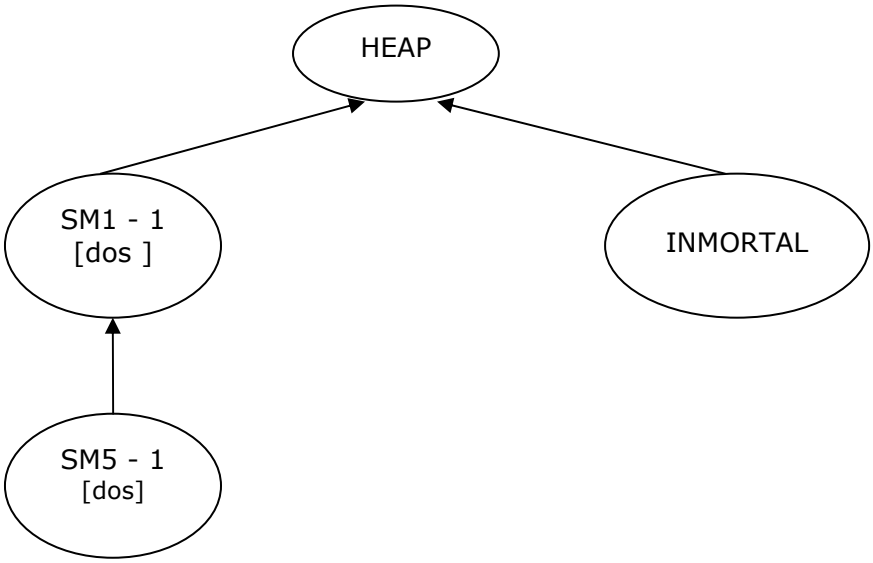


Figura 29. Recolección hilo 3

Finalmente eliminamos el hilo *dos*, quedando INMORTAL como único hijo del *HEAP*, habiéndose eliminado correctamente todos los hilos y todas las regiones *Scoped*.

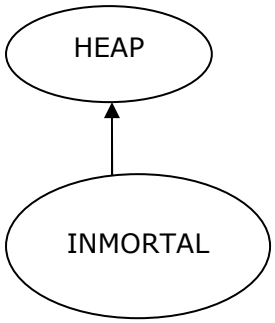


Figura 30. Recolección hilo 2

3.2.4.3.- Main

Esta clase es la principal de este modelo, donde se llamará a la interfaz para realizar el conjunto de acciones en base a las ínter actuaciones con el usuario y que básicamente contendrá un solo método:

public static void ***construyeSubarbol****(ArrayList graph)*

A medida que vamos creando el grafo completo generamos un subgrafo de todas las regiones *Scoped*, modificándose cuando eliminamos un hilo. Las relaciones existentes en el grafo inicial entre las regiones Scoped las seguirán manteniendo en este subgrafo, con la salvedad de que ya no influirán las posibles relaciones de parentesco que tenían con regiones Heap o Inmortal.

3.2.4.4.- Interfaz

La clase Interfaz, como ocurre en el resto de modelos propuestos es la encargada de la entrada y salida por pantalla y la ínter actuación con el usuario. Su estructura y características serán comentadas más abajo en la parte de pruebas indicando las partes de la interfaz y las funciones de cada una de sus botones.

Los atributos y métodos relativos a esta clase están todas relacionadas con las funciones de las distintas partes gráficas de la interfaz, siendo éstas las funciones de:

public void ***jButtonMataHilo_actionPerformed****(ActionEvent e)*: método encargado de realizar el recolectado de los MemoryAreas del hilo seleccionado por el usuario.

public void ***jButtonnuevoHilo_actionPerformed****(ActionEvent e):* encargado de realizar el conjunto de acciones necesarias para insertar un nuevo hilo de ejecución en el programa.

public void ***ponImagenenPanel****(Image m1,Image m2):* método encargado de actualizar las imágenes propias de la interfaz. Este mostrado se diferenciará de los otros modelos ya que en este caso hemos decidido mostrar en el panel de la izquierda de la pantalla el árbol general de Memory Areas; mientras que en la derecha se mostrará el subárbol de Scoped, mostrando únicamente las relaciones entre Memory Areas de este tipo del árbol general de la izquierda.

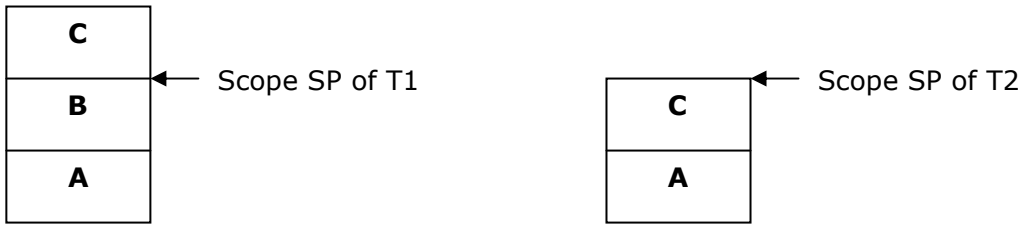
3.3.- Solución Alternativa.

Creación de regiones por varios hilos e interacción entre ellos

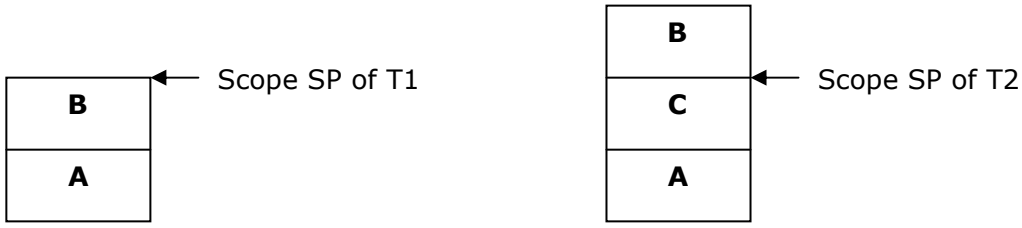
Para esta tercera simulación hemos tomado como referencia dos artículos de *M. Teresa Higuera Toledano* [2] y [3].

Consideremos tres áreas Scoped: A, B y C, y dos tareas T1 y T2. Donde la tarea T1 intenta entrar en las áreas como sigue: A, B y C, mientras T2 intenta entrar en las áreas en el siguiente orden: A, C y B. Vamos a suponer que la tarea T1 ha entrado en las áreas A y B, y la tarea T2 en las áreas A y C. Si la tarea T1 intenta entrar en el área C (*Figura 31.a*) o la tarea T2 intenta entrar en el área B (*Figura 31.b*), la regla del único padre es violada y como consecuencia se lanza una excepción *ScopedCycleException().*

Una desventaja de este modelo es el indeterminismo introducido por la regla del único padre.



a. T1 viola la regla del único padre



b. T2 viola la regla del único padre

Figura 31: Violación de la regla del único padre

Si por ejemplo, T2 entra en el área C antes que T1 intente entrar en ella, entonces es T2 quien viola la regla del único padre y provoca la excepción *ScopedCycleException()* (*Figura 32.a*). Si T1 entra en el área B antes que T2 intente entrar en ella, T2 viola la regla del único padre provocando la excepción *ScopedCycleException()* (*Figura 32.b*).

Hay que destacar que el determinismo es un requisito importante para las aplicaciones de tiempo real.

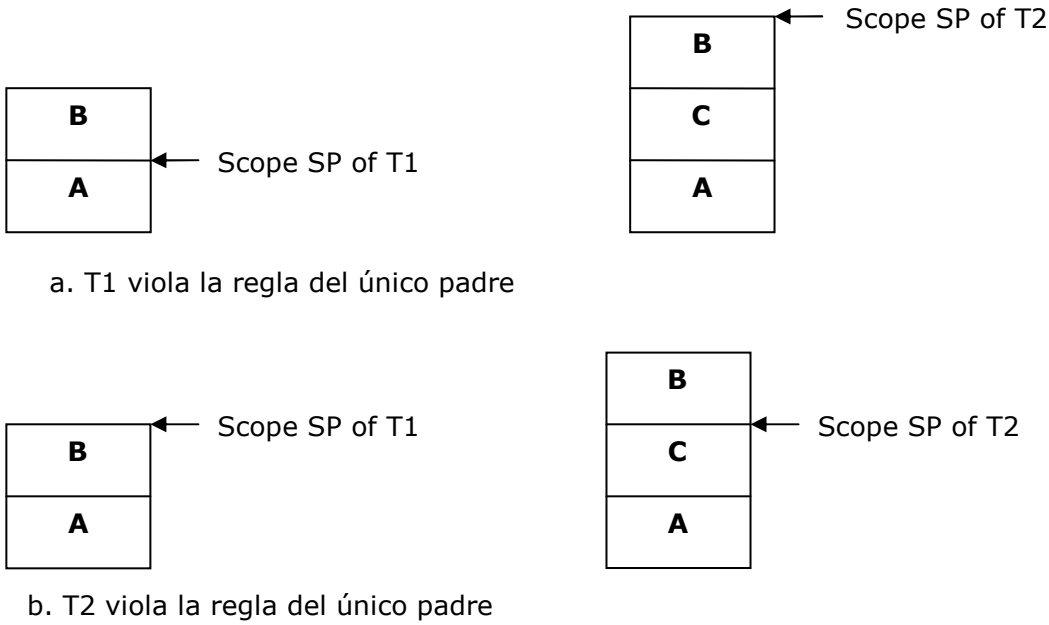


Figura 32: Ejemplo de una situación no determinista

Para solucionar el problema de indeterminismo introducido por las memorias *Scoped* en RTSJ necesitaremos redefinir la Regla del Único Padre. Esto es lo que haremos en el siguiente punto que mostramos como solución alternativa.

En este modelo al contrario que en el anterior se elude la exploración de la pila *Scoped*, sustituyendo ésta por una técnica basada en nombres forzando a que las referencias de memoria se hagan en tiempo predecible, es una solución orientada a la comprobación de las asignaciones ilegales. Este modelo propone basar la relación de parentesco de *Memory Áreas* en la forma, en el modo en que éstas son creadas/colectadas y se evita hacer comprobaciones cada vez que un *Scoped Memory* área intenta hacer un **enter**().

Las scoped memory áreas

En este modelo se considera que las regiones *Scoped* son recogidas por el recolector de basura como ocurría en el modelo anterior. El tiempo de vida de los objetos asignados en áreas *Scoped* está gobernado por el flujo control de flujo del programa, pero como no podemos similar este flujo, el usuario será el encargado de terminar el hilo a elección.

Una implementación para asegurar la comprobación de estas reglas antes de cada asignación consiste en llevarlas a cabo dinámicamente, cada momento una referencia se almacena en la memoria.

Regla del único padre

La relación de parentesco requiere que un área de memoria *Scoped* tenga exactamente cero o un padre. Los *Scoped* áreas que son entrados o propuestos como un área inicial de memoria para una nueva tarea deben satisfacer la regla del único padre.

La regla del único padre garantiza que un padre *Scoped* no tendrá un tiempo de vida menor que cualquiera de sus hijos, lo que hará seguras las referencias desde objetos en una región *Scoped* dada a objetos en una antecesora.

3.3.1.- La solución basada en nombres

En el actual RTSJ, cuando una tarea o un evento manejador intenta entrar en un área *Scoped* S, debemos comprobar si el correspondiente hilo ha entrado en alguno de los antecesores del área S en el árbol de áreas *Scoped*. Los test requieren algoritmos, cuyo coste es lineal o polinomial en el número de áreas de memoria que las tareas pueden mantener. Para optimizar el modelo de memoria de RTSJ este nuevo modelo sugiere simplificaciones de estructuras de datos y algoritmos y define cambiar la definición de la regla del único padre.

El indeterminismo de la regla del único padre

Como hemos comentado anteriormente, la regla del único padre introduce un cierto grado de indeterminismo, ya que garantiza que una vez ha entrado un hilo en un área *Scoped* en un orden dado, cualquier otro hilo es forzado a entrar en las áreas en el mismo orden.

La relación de parentesco propuesta

Para solucionar el problema de indeterminismo introducido por las memorias *Scoped* en RTSJ, se redefine la regla del único padre como sigue:

"*El padre de un área Scoped es el área en el cual el objeto que representa el área Scoped es creado."*
Entonces, se propone basar la relación de parentesco en la forma en que las áreas *Scoped* son creadas, en lugar de en el orden en el que las áreas *Scoped* han sido entradas por hilos como en RTSJ. Para ello hay que tomar en cuenta las siguientes modificaciones:

i) La relación de parentesco de áreas implica mantener solamente una estructura de árbol *Scoped*, que es compartido por todos los hilos de tiempo real de las aplicaciones; en lugar de mantener una pila de *Scoped* por cada hilo de tiempo real, como sugiere RTSJ.

ii) La clase *Scoped Memory* contiene el método *getOuterScope()*, que nos permite conocer para la tarea actual el antecesor del área de memoria entrado actualmente.

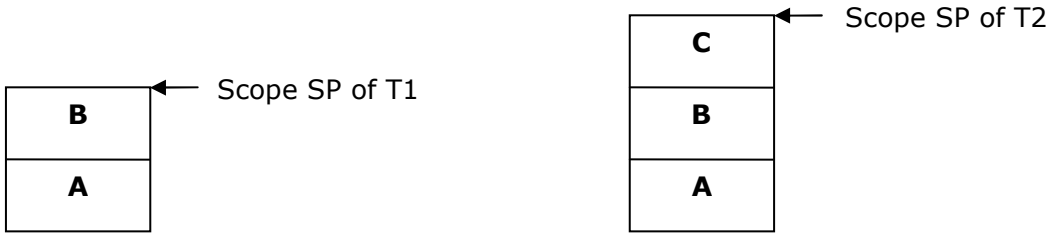
iii) Cada instancia de la clase *ScopedMemory* o sus subclases debe mantener un contador de referencias con el número de hilos de tiempo real contenidos en el área actual *(contador de tareas)*, y también un contador de referencias con el número de *Scoped* áreas creadas dentro del área *(contador de hijos)*. En la actual especificación de RTSJ se mantiene un contador de referencias para los hilos de tiempo real usando el *Scoped* área, y otro para el numero de hijos que posea. Cuando ambos contadores llegan a cero el área *Scoped* es un candidato a ser recuperado.

El determinismo de la solución propuesta

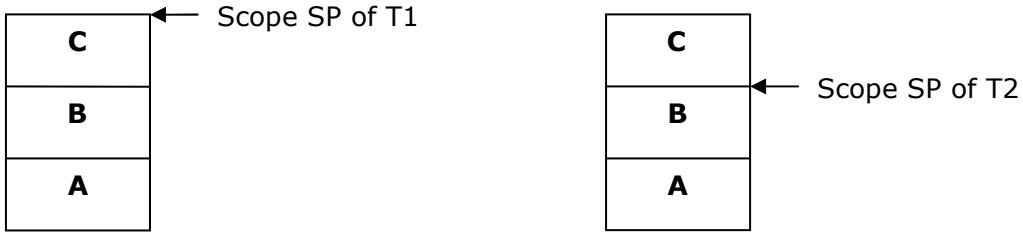
Consideremos tres áreas *Scoped*: A, B y C, que han sido creadas de la siguiente forma: el área A ha sido creada dentro del *Heap*, el área B ha sido creada dentro del área A y el área C ha sido creada dentro del área B. Eso quiere decir que el *Heap* era el área actual cuando se creó el objeto A, A era el área actual cuando se creó el objeto B, y B era el área actual cuando se creó el objeto C. De este modo, la creación de las áreas *Scoped* A, B y C viene dada por la siguiente relación de parentesco: el *Heap* es el padre de A, el área A es el padre de B, y B es el padre de C. Entonces, el contador de hijos para Ay B ha sido incrementado en uno, mientras que el de C se mantiene en cero.

Consideremos las tareas T1 y T2 del ejemplo anterior, donde hemos supuesto que la tarea T1 ha entrado en las áreas A y B, y que se incrementa el contador de tareas en uno para A y B. La tarea T2 ha entrado en los áreas A y C, se incrementa el contador de tareas en uno para A y C (*Figura 33.a*). En esta situación el contador de tareas para A es dos, mientras que para B y C es uno. Si T1 entra en el área C y T2 en el área B, a diferencia de cómo ocurría en RTSJ, la regla del único padre no es violada. Entonces, en vez de lanzarse una excepción *ScopedCycleException(),* tenemos una situación como la mostrada

en la *Figura 33.b*. En este momento el contador de tareas para los áreas de memoria *Scoped* A, B y C es dos.



a. T1 entra en el área scoped B y T2 entra en C

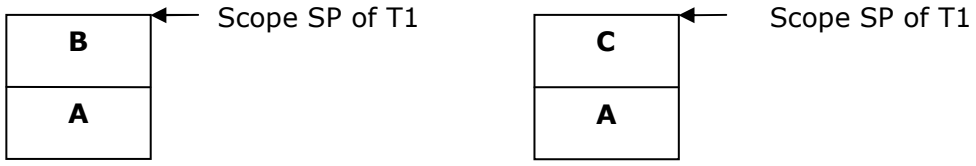


b. T1 entra en el área scoped C y T2 entra en B

Figura 33: La pila scope y la regla del único padre.

Notar que la pila de *Scoped* asociada a la tarea T2 incluye sólo las áreas *Scoped* A y B. Si la tarea T2 ha entrado en el *Scoped Memory* C antes de entrar en B, los punteros de los objetos alojados en B hacia objetos alojados en C son punteros colgados, como consecuencia éstos no son permitidos.

Consideramos otra situación: La tarea T1 entra en el área *Scoped* A y crea B y C, que incrementa su contador de tareas en uno y su contador de hijos en dos. Los contadores de tareas y de hijos de B y C son cero. Entonces, la tarea T1 entra en los áreas *Scoped* B (*Figura 34.a*) y C (*Figura 34.b*), incrementando el contador de tareas en uno, tanto para B como para C. En esta situación, sólo las referencias desde objetos alojados dentro de B o C a objetos dentro de A son permitidas. No es posible para la tarea T1 crear una referencia desde un objeto dentro de B a un objeto dentro de C, y viceversa desde un objeto dentro de C a un objeto dentro de B, incluso si la tarea T1 debe salir del área C antes que del área B. Si la tarea T2 entra en el área *Scoped* C y permanece allí mientras la tarea T1 deja C y B, el *Scoped* área B puede ser colectado y no habrá punteros colgados.



a. T1 entra en el área B b. T1 entra en el área C

Figura 34: Dos estados para la pila scoped para la tarea T1.

Heap e *Inmortal* áreas no son soportadas en el árbol *Scoped*. Estas áreas son consideradas como el *primordial Scoped,* que son consideradas la raíz del árbol *Scoped*. Tanto el *Heap* como el *Inmortal Memory* no necesitan mantener un contador de referencias.

Comprobando las reglas de asignación

En la implementación de este nuevo modelo propuesto la relación de parentesco cambia en determinados momentos, es decir, cuando se crean o colectan áreas de memoria.

RTSJ impone en las reglas de asignación, que las referencias pueden siempre ser hechas desde objetos dentro de un *Scoped Memory* a objetos dentro de *Heap* o *Inmortal Memory*; lo opuesto nunca está permitido.

Consideremos dos áreas de memoria, A y B, donde A es padre de B. Una referencia al *Scoped* área A puede ser referenciada desde un campo de un objeto alojado en B. Pero una referencia desde un campo de un objeto dentro de A a otro objeto alojado dentro de B provoca una excepción *IllegalAssigment().*

Cada objeto área de memoria *Scoped* debe mantener un contador de referencias con el número de hilos por el que está siendo usado. Cuando este contador de referencias se decrementa a cero, todos los objetos alojados dentro de este área de memoria son considerados inalcanzables y son candidatos a una recuperación.

La administración de nombres de áreas de memoria sólo requiere una copia del nombre del padre e incluir un nuevo identificador del área creado al final para cuando creamos un área *Scoped*.

Se propone una técnica basada en nombres que realiza las comprobaciones para las reglas de asignación en tiempo constante.

Consideramos tres áreas *Scoped*: A, B y C, con la siguiente relación de parentesco: el *Heap* es el padre de A, el área A es el padre de B, y B es el padre de C. Entonces, el nombre del área A es **'A',** el nombre del área B es **'AB'**, y el nombre del área C es **'ABC'** (*Figura 35*).

Esta relación de parentesco es menos dinámica que la de la implementación anterior, donde la relación padre-hijo cambia cuando un *Scoped* área de memoria recibe un ***enter***() o ***exit***(). En esta nueva propuesta la relación padre-hijo sólo cambia cuando un área de memoria *Scoped* es creado o destruido, es decir, cuando el contador de referencias de hijos se incrementa o decrementa. La estructura del árbol *Scoped* no se ve afectada cuando entramos/salimos de un área de memoria sino cuando creamos/destruimos un hilo.

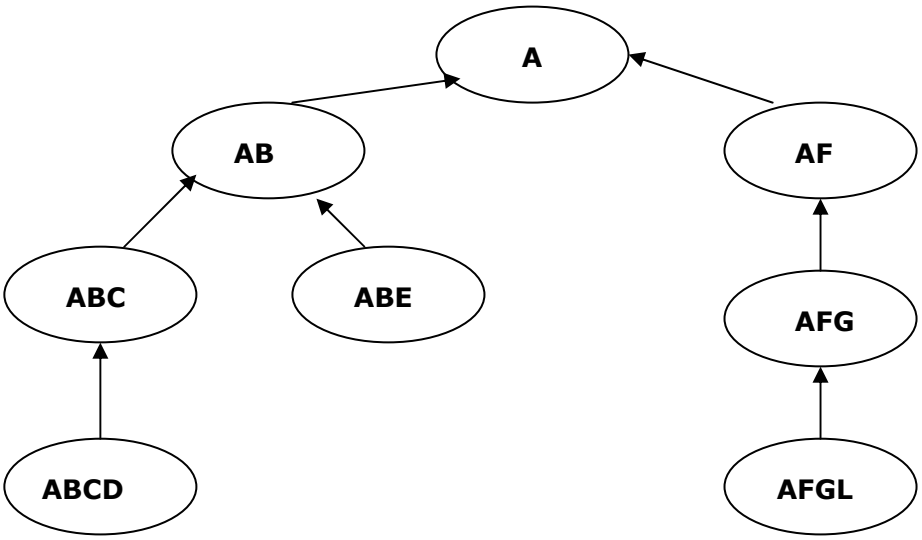


Figura 35: Estructura del árbol de Memory area.

La *Figura 36* muestra el pseudo-código que debemos introducir a cada asignación para comprobar las asignaciones en tiempo constante.

```
X = nombre de la región a la cual pertenece el objeto x
Y = nombre de la región a la cual pertenece el objeto y

Si (( Y && X) <> Y ) illegalAssignment();
```

Figura 36: Write Barrier.

Esta nueva relación de parentesco introduce grandes avances:

1. Simplifica la semántica del *Scoped Memory* como que la regla del único padre llegue a ser siempre cierta.

2. No saltan excepciones de ciclos.

3. La relación de parentesco no cambia durante la vida del *Scoped*.

Las regiones que no pertenecen al *Scoped*, es decir, *Heap* e *Inmortal* no están soportadas por el árbol. Además, no necesitan contadores de referencias porque su existencia está fuera del ámbito de la aplicación.

Igual que en el modelo anterior teníamos el método *getReferenceCount*() para saber cuantas referencias tenía, ahora se introducen dos métodos nuevos: *getChildrenCount*() y *getTaskCount*() que nos permiten saber respectivamente el número de regiones *Scoped* creadas dentro y el número de tareas.

- Cuando una región *Scoped* llega a ser la región actual para una tarea (p. ej.: entrando en ella o creando un hilo de tiempo-real) incrementamos el contador de referencias de tareas de la región. Y lo decrementamos cuando la tarea deja la región (a la salida del método ***enter***()).

- Cuando se crea un nueva región *Scoped*, aumenta el contador de referencias de hijos del padre, y debemos decrementarlo cuando la nueva región es recolectada. Los dos contadores de la nueva región se inicializan a 0, y cuando se recicla hay que comprobar que ambos están a 0.

Esta solución requiere de una sola acción (incrementar/decrementar un contador) cuando una tarea es creada o destruida, o cuando una nueva región entra o sale.

Mantener la pila del *Scoped*

En la edición primera de la implementación el método ***enter***() puede lanzar la excepción *ScopeCycleException*() además se chequean con la regla del único padre las relaciones entre las regiones.

En la solución que se propone no hay ciclos en la estructura del árbol y se evita el chequeo con la regla del único padre. También se considera que el árbol contiene todas las posibles pilas de *Scoped* para todas las tareas de la aplicación en un determinado momento. Donde cada pila de *Scoped* está compuesta por la región actual y por todas las regiones siguientes en la ruta para alcanzar la raíz del árbol. El puntero a la pila del *Scoped* es parte del contexto de ejecución de la tarea. Se muestra a continuación el pseudo-código para la operación ***enter***(), con tiempo constante de ejecución.

3.3.2.- Conclusiones

Para hacer cumplir las reglas impuestas del RTSJ, la JVM debe chequear la regla del único padre en cada intento para entrar en una región de *Scoped Memory*, y en las reglas de asignación en cada intento de creación de una referencia entre objetos que se encuentran en distintas regiones de memoria. Como las referencias a objetos ocurren muy a menudo, es importante implementar chequeos para las reglas de asignación que sean eficientes y predecibles.

Para soportar las regiones de memoria, se propone un mecanismo basado en un colector de contadores de referencia y en un árbol de regiones *Scoped* basado en la relación de parentesco de las regiones, que contiene todas las pilas de *Scoped* que permite el sistema en un momento dado. A la hora de reciclar las regiones, los problemas asociados con los colectores de los contadores de referencias están resueltos: el espacio y el tiempo para mantener dos contadores de referencias por *Scoped* es mínimo, y no se producen ciclos en las referencias entre las regiones.

La relación de parentesco propuesta para las áreas de memoria permite usar una técnica basada en nombres para comprobar las referencias ilegales, lo cual simplifica la implementación sugerida por RTSJ (implementaciones primera y segunda) basada en una pila de *Scoped*. Desde que las comprobaciones de referencias ilegales requieren acciones antes de cada asignación, que afecta de forma adversa tanto a la predecibilidad como a la representación de la aplicación RTSJ, esta relación de parentesco sugerida resulta particularmente interesante.

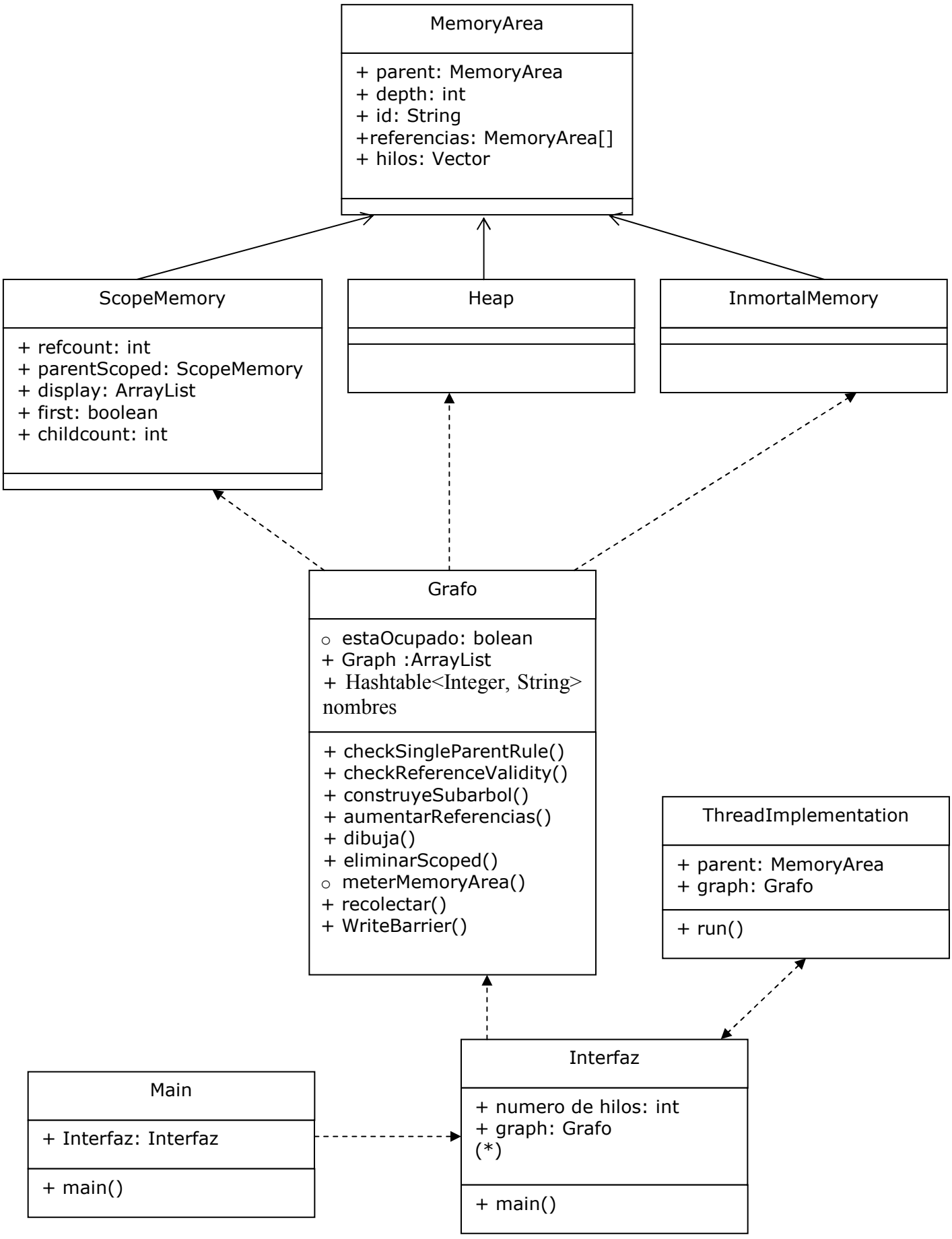
La solución propuesta requiere que todas las áreas *Scoped* tengan dos contadores de referencia asociados. Para colectar áreas, los problemas asociados con los contadores de referencias son solventados, el espacio y el tiempo para mantener dos contadores de referencias por *Scoped* área es mínimo, y no hay referencias cíclicas a *Scoped Area*.

La introducción de este cambio en la relación de parentesco simplifica la complejidad semántica de las áreas de memoria *Scoped* adoptadas por RTSJ.

3.3.3.- Diagrama de interacción entre clases

Este diagrama de clases va a ser similar al de la primera implementación basada en hilos con el fin de mantener una estructura uniforme entre las distintas alternativas.

En este nuevo modelo ha sido necesario introducir o modificar algunos métodos a fin de lograr los objetivos planteados en esta sección. Figura más abajo


MemoryArea
+ parent: MemoryArea
+ depth: int
+ id: String
+referencias: MemoryArea[]
+ hilos: Vector
ScopeMemory
+ refcount: int
+ parentScoped: ScopeMemory
+ display: ArrayList
+ first: boolean
+ childcount: int
Heap
InmortalMemory
Grafo
○ estaOcupado: bolean
+ Graph :ArrayList
+ Hashtable<Integer, String> nombres
+ checkSingleParentRule()
+ checkReferenceValidity()
+ construyeSubarbol()
+ aumentarReferencias()
+ dibuja()
+ eliminarScoped()
○ meterMemoryArea()
+ recolectar()
+ WriteBarrier()
ThreadImplementation
+ parent: MemoryArea
+ graph: Grafo
+ run()
Main
+ Interfaz: Interfaz
+ main()
Interfaz
+ numero de hilos: int
+ graph: Grafo
(*)
+ main()

3.3.4.- Implementación

Mediante un conjunto de hilos introducido por el usuario se empezarán a generar regiones de memoria de tipo *Scoped, Heap o Inmortal.* La principal diferencia con el modelo anterior es que en este caso se genera la relación de parentesco en el momento de la creación de la región, su padre será la región dentro de la que ha sido creada.

Otra característica introducida en esta implementación es un contador en la clase de *Scoped Memory* para llevar la cuenta del número de hijos de cada Scoped *Memory* en todo momento, además del contador de tareas que ya teníamos en la solución anterior.

A continuación pasamos a detallar como hemos llevado a cabo el desarrollo para esta implementación comentando cada una de las clases con sus atributos y la relación que guardan entre ellas.

3.3.4.1.- Memory Area

Es la clase de la que heredan *Inmortal Memory*, *Heap Memory* y las *Scope Memory*.

Para poder llevar a cabo la implementación del modelo los objetos de la clase Memory Area necesitarán tener los siguientes atributos:

Atributos:

private MemoryArea parent;

Necesitamos este atributo de tipo *MemoryArea* denominado *parent* para referirnos al padre de cada región creada.

private int depth;

Poseerá también un atributo de naturaleza entera *depth* que indicará la profundidad del área *Scoped* dentro del árbol de referencias.

private String id;

Hemos capacitado a esta clase con un identificador de objeto, de tipo *String* que nos ayudará a identificar cada *MemoryArea*.

private Vector hilos;

El atributo *hilos* es un Vector que contiene el identificador de los hilos que han entrado en el área de memoria.

private MemoryArea[] referencias;

El atributo MemoryArea[] *referencias* nos ayuda a simular las asignaciones entre *Memory Areas*, en principio hemos puesto un número fijo de referencias a cada *Memory Area*, hemos elegido 3. El número de referencias se refiere a referencias entre regiones, es una abstracción del modelo *RTSJ* para estudiar el comportamiento del mismo.

Constructor

public **MemoryArea**()

Generamos un nuevo objeto de tipo *MemoryArea*. Inicializamos la profundidad de cada *MemoryArea* a –1 ya que asumimos que tanto *Heap* como *Inmortal* tienen profundidad -1, el padre a null, las referencias a null y el vector de hilos vacio.

Métodos

Hemos definidos tanto los métodos accesores como los mutadores de cada atributo de un *MemoryArea*.

3.3.4.2.- Heap Memory

Extiende de la clase *MemoryArea*, heredando todos sus atributos.

3.3.4.3.- Inmortal Memory

Extiende de la clase *MemoryArea*, heredando todos sus atributos.

3.3.4.4.- Scope Memory

Extiende de la clase *MemoryArea*, heredando todos sus atributos.

En relación a las *ScopeMemory*, hemos de garantizar una propiedad básica; la regla del único padre (*Single Parent Rule*), garantizando que cada *Scoped Memory* tiene al menos un padre y sólo será ese durante toda su vida.

Por otro se comprobará que las referencias sean válidasmediante reglas de asignación basadas en los nombres de las regiones.

Comentaremos con más detalle estas funciones en la clase *Grafo*, que es donde están implementadas.

Atributos

private int refcount;

El atributo *refcount* de naturaleza entera llevará la cuenta de los hilos que han entrado en esa región. Se incrementará cuando el hilo entre en la región.

private int childcount;

El atributo *childcount* de naturaleza entera llevará la cuenta de los hijos que tiene cada *Scoped Memory*. Se incrementará cada vez que se cree una región *Scoped* dentro de otra.

Los objetos serán recolectados cuando ambos contadores alcancen el valor 0.

private ScopeMemory parentScoped;

Es un puntero al padre de tipo *Scoped Memory* de este objeto. Asignamos este puntero con la generación de un árbol cuyos nodos son sólo regiones de tipo *Scoped.*

private ArrayList display=new ArrayList() ;

Guardamos en un array los antecesores en el árbol; este display sólo contendrá regiones de tipo *Scoped Memory*.

Constructor

public **ScopeMemory**()

Crearemos un objeto de tipo *Scoped Memory* inicializando *refcount* a 0, ya que inicialmente no hay ninguna referencia a esta región, *childcount* a 0, *parentScoped* a null, el display de regiones de memoria de tipo Scoped vacío y *first* a false.

Métodos

Accesores y mutadores de los atributos de la clase.

3.3.4.5.- Grafo

Para llevar a cabo la implementación esta clase cuenta con un número elevado de atributos, entre los que destacan:

Atributos

private ArrayList graph;

El atributo *graph* de tipo array va a ser el contenedor para guardar todas las regiones de memoria que se van creando. Al inicio de la implementación sólo tendrá un elemento, el *HEAP* o *PimordialScope*, a medida que los hilos crean nuevas regiones se irán añadiendo a este array.

El atributo *graph* va a comportarse como una región crítica, es decir, mientras un hilo está utilizando el array (un hilo ha entrado en una de las regiones que contiene el array) éste estará inaccesible para el resto, ya que podrían darse situaciones inconsistentes o de conflicto. Para que el resto de hilos no puedan acceder a la región crítica mientras uno de ellos está dentro utilizamos este atributo *estaOcupado* como flag para saber el estado del array. Lo inicializamos a *false* indicando que al inicio no hay ningún hilo utilizando la región crítica.

private boolean estaOcupado;

Constructor

public **ScopeMemory**()

Mete en el array *graph* inicialmente vacío el primer elemento *HEAP* o *PrimordialScope*.

Métodos

Además de los métodos accesores y mutadores de los atributos tenemos los siguientes:

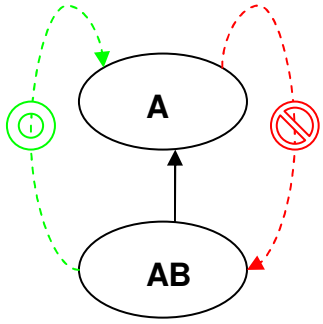
*public void **aumentarReferencias**(ScopeMemory ma,String nombre)*

Dado un área de memoria *ma* y un hilo aumento en 1 el contador de referencias de *ma* si ha sido entrado por el hilo identificado por su nombre.

*public boolean **WriteBarrier**(MemoryArea padre, String hijo)*

Con este método comprobamos la validez de las reglas de asignación en tiempo constante. La asignación será legal siempre que se haga desde un objeto de un *Scoped Memory* a objetos dentro de *Heap* o *Inmortal*, o siempre que se haga desde un objeto de un *Scoped Memory* a otro objeto de un *Scoped Memory* antecesor suyo.

La Figura 36 muestra el pseudo código que hace posible la comprobación de las reglas de asignación en tiempo constante.
La Figura 37 muestra un ejemplo de una asignación legal, ya que se hace desde un objeto del hijo AB a un objeto del padre A. La Figura 37 muestra una asignación ilegal desde un objeto de un padre A a un objeto alojado en una región hija AB.



AB = nombre de la región a la cual pertenece el objeto ab
A = nombre de la región a la cual pertenece el objeto a
Si ((A && AB) <> A) entonces IllegalAssignment()

Como el resultado de la operación (A && AB) es A la asignación entre objetos es legal, en caso contrario será ilegal como indica la flecha roja

Figura 37. Asignaciones ilegales

public synchronized int ***meterMemoryArea*** *(MemoryArea nueva, Boolean crear, String nombre)*

Cada vez que un hilo intenta entrar en una región debe comprobar antes que la región crítica no esta siendo usada por otro hilo, si es así tendrá que esperar hasta que quede libre.

El parámetro *crear* indica si el hilo va a crear una nueva región *Scoped* o si solamente va a entrar en ella. Si estamos en el primer caso identificaremos *nueva* con el nombre de su padre, región en la que estaba el hilo cuando creo el nuevo área, concatenado de su propio nombre. Aumentaremos en 1 el contador de hijos del padre y aumentaremos en 1 el contador de referencias del hijo utilizando la función anteriormente descrita *aumentarReferencias.* Si estamos en el segundo caso, el hilo entra en una región ya creada, al igual que antes aumentaremos en 1 el contador de referencias del hijo utilizando la función anteriormente descrita *aumentarReferencias.*

public void ***eliminarScope*** *(ScopeMemory sm, String hilo)*

Elimina de la estructura del árbol aquellos *Scoped* que tienen el valor 0 tanto en el contador de referencias como en el contador de hijos.

public void ***recolectar****(String hilo)*

Cuando matamos un hilo, que se puede hacer desde la interfaz como se verá más adelante, actualizamos el contador de referencias de los *Scoped* en los que había entrado el hilo decrementando en 1 su valor. Si al decrementar en 1

este valor llegamos a que ambos contadores tanto el de referencias como el de hijos valen 0 eliminaremos el Scoped con ayuda de la función *eliminarScope*, y tendremos que actualizar el contador de hijos de los padres decrementando en 1 su valor.

public void ***chequeaReferencias****()*

Al igual que en las dos anteriores implementaciones asignamos de forma aleatoria referencias a otras 3 regiones *Scoped*, en este caso de forma distinta a las otras implementaciones utilizamos el método *WriteBarrier* explicado anteriormente para comprobar la validez de las asignaciones. Metemos como parámetros del método *WriteBarrier* el *Scoped* en el que está el objeto origen y el *Scoped* en el que está el objeto referenciado.

3.3.4.6.- Thread Implementation

Los hilos se crearán a petición del usuario y éstos podrán crear o referenciar regiones de memoria. Se ha creado un estándar para los hilos, de tal forma, que todos creen o referencien 10 veces (aunque se puede modificar). Para decidir si crean o referencian tenemos una variable que usa el *random* y que concede la misma prioridad a las dos acciones.

- *Creación*: El hilo tiene otro random para decidir qué tipo de región de memoria va a crear: *Heap*, *Inmortal* o *Scope*, siendo el *Scope* el que tiene mayor prioridad, aunque casi imperceptible. Una vez decidido, el hilo entra en la región crítica y le asigna un nombre a la región nueva. Lo inserta en el grafo utilizando las funciones propias del ArrayList, le asigna el padre y aumenta las referencias en caso de que la nueva región sea *Scope*.

- *Referenciación*: El hilo entra en la región crítica en la zona de referencias. En esta parte del código el hilo busca una región, usando la regla del único padre, a la que pueda referenciar. Si la región que va a referenciar es *Scope* tiene que aumentar el contador de referencias de esa región si este hilo no la había referenciado antes.

Cuando el hilo ha terminado de hacer todas sus operaciones permanece latente mientras el usuario no diga lo contrario. Igual que en la creación, el hilo morirá a petición del usuario.

3.3.4.7.- Main

Inicia la ejecución de la implementación lanzando la interfaz.

3.3.4.8.- Interfaz

La estructura de la interfaz es la misma que en los otros modelos, poseerá el botón de insertar un nuevo hilo de ejecución, y el botón de eliminar el hilo seleccionado en el combo box de hilos de ejecución.

La diferencia con las otras implementaciones es que en este caso sólo nos vamos a fijar en la estructura y las relaciones existentes entre regiones de memoria de tipo *Scoped*.

De esta forma se generará en cada inserción un árbol de *Scoped Memory*, colgadas todas de un padre en común, el *Heap*, y poseyendo cada una lista de los hilos que han entrado en las regiones y el número de hijos que posee esa *Scoped*, y una lista de los hilos que han entrado en esa área, como se muestra en la Figura 38.

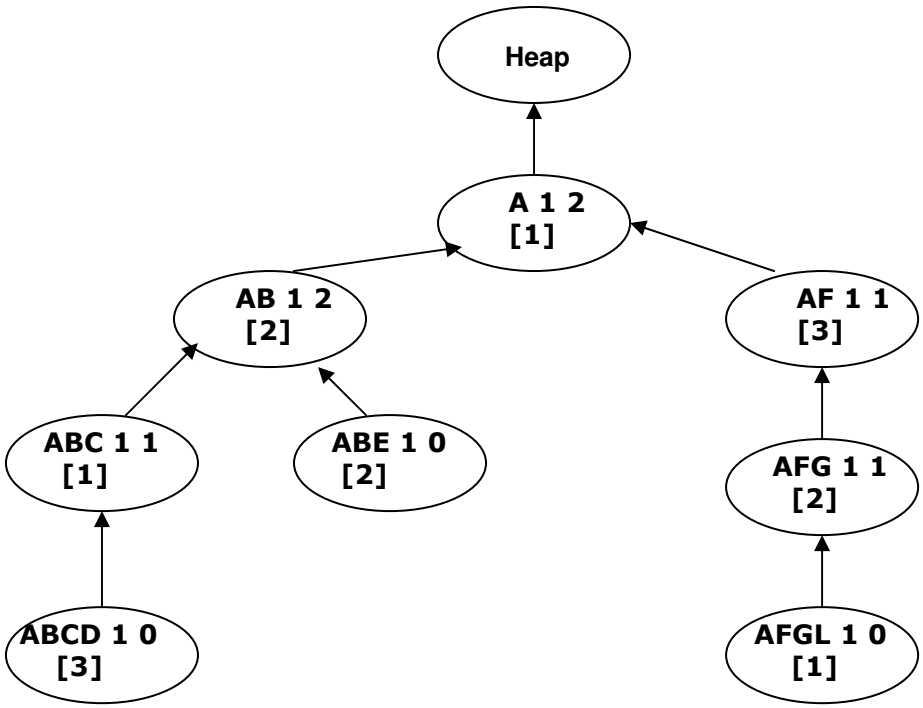


Figura 38. Árbol ejemplo nombres

Otra novedad en la interfaz es que vamos a llevar un grafo con el movimiento realizado anterior por lo tanto:

- Ante una inserción se presentará en el árbol del lado izquierdo la estructura del grafo sin la última inserción de una *Memory Area*

- Ante la eliminación de un hilo de ejecución en el panel del lado izquierdo estará el grafo antes de que el hilo, y por lo tanto las posibles regiones *Scoped* sean recolectadas; mientras que en el panel del lado derecho mostraremos el árbol actualizado tras la eliminación del hilo seleccionado por el usuario. De esta forma se verá fácilmente que ha sucedido en el árbol de *Scoped* tras esa eliminación.

3.3.4.9.- Ejemplo Aplicación

Creamos dos hilos y obtenemos el siguiente árbol de parentesco (Figura 39):

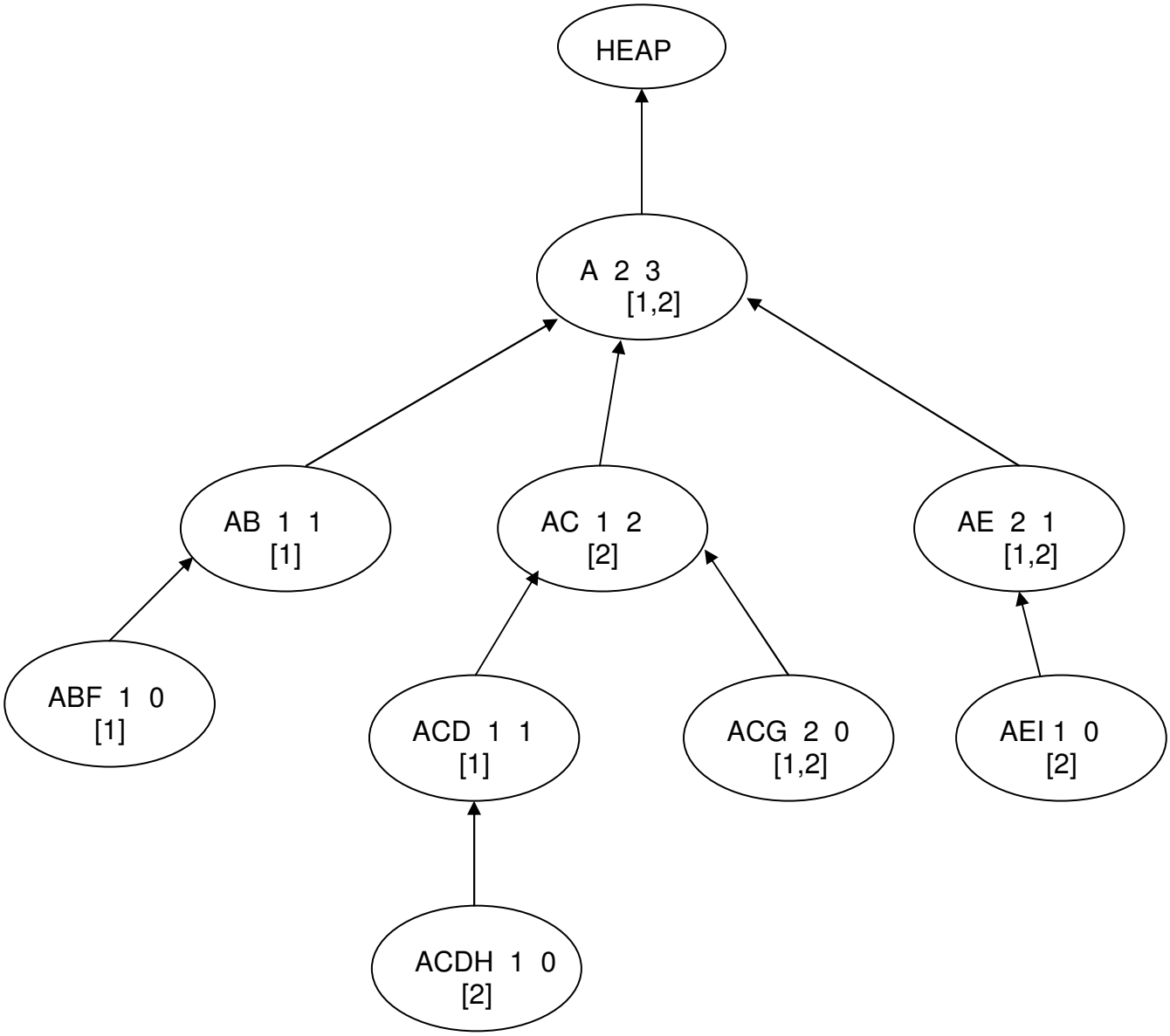


Figura 39. Introducción hilos 1 y 2 nombres

Cada nodo del árbol muestra el número de hilos que le han entrado y el número de hijos que tiene.
El Hilo 1 crea el Scoped A en el HEAP, de ahí que éste sea su padre.
El Hilo 1 entra en A, aumentando el contador de referencias de la región en 1.
Estando en A el hilo crea AB y AE, aumentando el contador de hijos en 2.
El Hilo 2 entra en A, aumentando el contador de referencias de la región en 1, con lo que el contador de referencias pasará a valer 2. Estando en A el hilo crea AC aumentando el contador en 1, ahora pasará a valer 3.
El Hilo 1 entra en AB, aumentando el contador de referencias de la región en 1.
Estando en AB el hilo crea ABF aumentando el contador de hijos en 1.
El Hilo 1 entra en ABF aumentando en 1 el contador de referencias.
El Hilo 2 entra en AC, aumentando el contador de referencias de la región en 1.
Estando en AC el hilo crea ACD y ACG aumentando el contador de hijos en 2.
Los Hilos 1 y 2 entran en ACG aumentando en 2 el contador de referencias.
El Hilo 1 entra en ACD aumentando el contador de referencias en 1, estando en esta región el hilo crea ACDH aumentando el contador de hijos en 1.
El Hilo 2 entra en la región ACDH aumentando el contador de referencias en 1.
Los Hilos 1 y 2 entran en AE incrementando el contador de referencias en 2.
Estando en AE el hilo 2 crea AEI aumentando el contador de hijos en 1.
El Hilo 2 entra en AEI aumentando el contador de referencias en 1.

Redefinimos la Regla del Único Padre respecto a las implementaciones anteriores. Basamos la relación de parentesco en la forma en que las áreas *Scoped* son creadas, el padre de una región es el área donde la hemos creado. El modelo anterior fijaba el padre de cada *Scoped* mediante el método que implementa la Regla del Único Padre en función del orden en el que las áreas Scoped habían sido entradas por los hilos.

A continuación matamos el hilo 1, decrementando los contadores de forma adecuada (Figura 40):

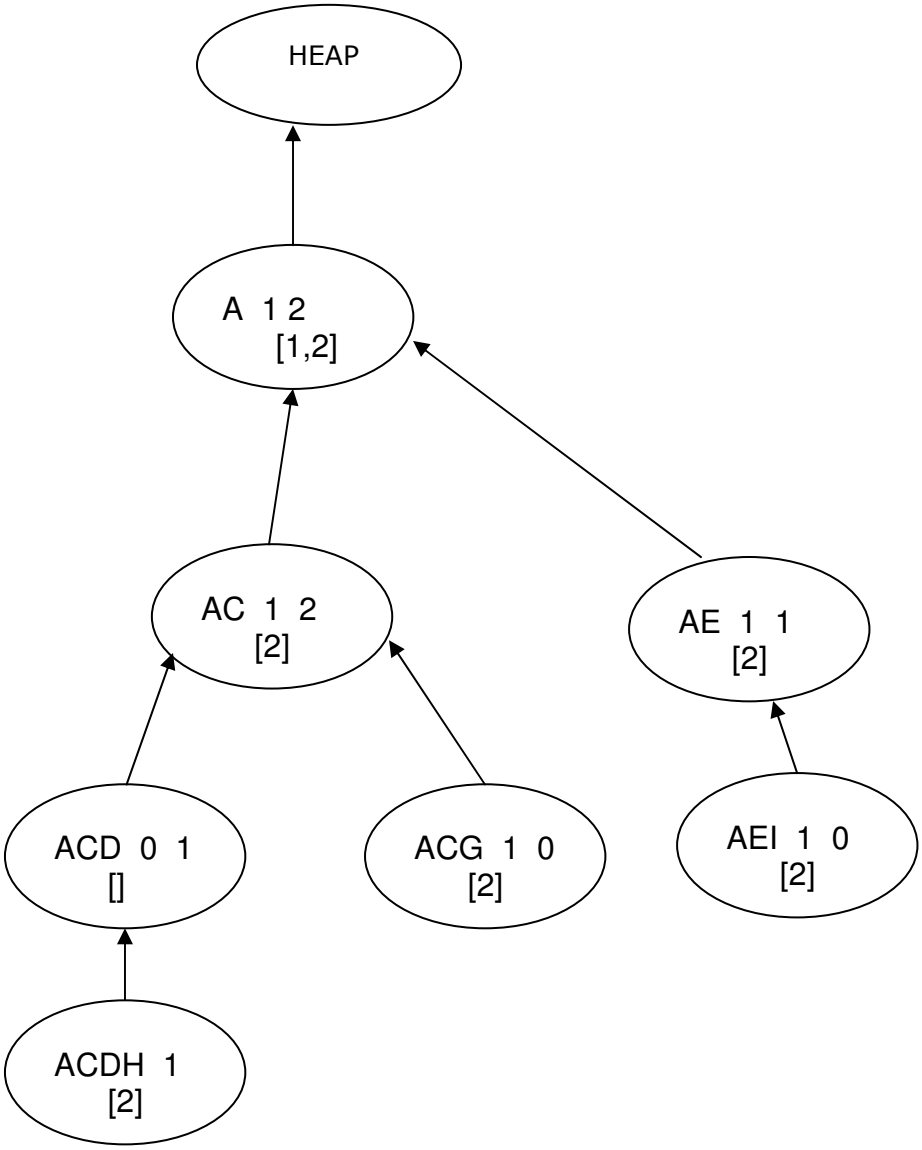


Figura 40. Recolección hilo 1 nombres

En todos los nodos en los que hubiese entrado el Hilo 1 se decrementa en 1 el contador de referencias. Si algún nodo hoja había sido entrado sólo por le Hilo 1 habrá sido recolectado, ya que tanto su contador de referencias como su contador de hijos valdrán 0, y esto se propagará hacia sus padres. Por ejemplo el nodo hoja ABF pasa a tener 0 hilos y 0 hijos, se recolecta. Al recolectarse su padre AB pasa a tener también 0 hilos (ya que matamos el Hilo 1) y 0 hijos (ya que su único hijo ABF ha sido recolectado), con lo cual también se recolecta.

Si a continuación matamos el hilo 2 el árbol quedará vacío. Se empezará a recolectar por los nodos hoja y se propagará hacia los padres, de igual que al matar el hilo 1.

Por lo tanto las condiciones necesarias para poder llevar a cabo la recolección de una región de memoria de tipo *Scoped* en este modelo son:

- Que su contador de hilos esté a 0, es decir, ya no exista ningún hilo de tiempo real dentro de ella

- Su contador de hijos esté a 0, y por lo tanto no cuelgue ninguna región de memoria de ella.

Como parte del ejemplo de aplicación se comprobaría si las referencias de un objeto a otro son válidas. Como hemos comentado anteriormente para comprobar si las asignaciones son legales utilizamos la función WriteBarrier. Cualquier asignación de un objeto de un hijo aun objeto contenido en un padre será legal.

En el Texto 4 mostramos un ejemplo para el chequeo de referencias en esta implementación, haciendo uso de *WriteBarrier*:

```
Chequear referencias válidas del nodo : HEAP al nodo HEAP
true
Chequear referencias válidas del nodo : HEAP al nodo ABIK
false
Chequear referencias válidas del nodo : HEAP al nodo ABCJ
false

Chequear referencias válidas del nodo : A al nodo ABEHL
true
Chequear referencias válidas del nodo : A al nodo AB
true
Chequear referencias válidas del nodo : A al nodo ABCG
true

Chequear referencias válidas del nodo : AB al nodo AF
false
Chequear referencias válidas del nodo : AB al nodo ABEHL
true
Chequear referencias válidas del nodo : AB al nodo ABCJ
true

Chequear referencias válidas del nodo : ABC al nodo AF
false
Chequear referencias válidas del nodo : ABC al nodo ABCG
true
Chequear referencias válidas del nodo : ABC al nodo ABIK
false

Chequear referencias válidas del nodo : ABCD al nodo AB
false
Chequear referencias válidas del nodo : ABCD al nodo ABI
false
Chequear referencias válidas del nodo : ABCD al nodo ABCG
false

Chequear referencias válidas del nodo : ABE al nodo A
false
Chequear referencias válidas del nodo : ABE al nodo ABCG
false
```

```
Chequear referencias válidas del nodo : ABE al nodo ABEHL
true

Chequear referencias válidas del nodo : AF al nodo AF
true
Chequear referencias válidas del nodo : AF al nodo A
false
Chequear referencias válidas del nodo : AF al nodo ABCJ
false

Chequear referencias válidas del nodo : ABCG al nodo ABCJ
false
Chequear referencias válidas del nodo : ABCG al nodo AB
false
Chequear referencias válidas del nodo : ABCG al nodo ABC
false

Chequear referencias válidas del nodo : ABEH al nodo AB
false
Chequear referencias válidas del nodo : ABEH al nodo AF
false
Chequear referencias válidas del nodo : ABEH al nodo ABIK
false

Chequear referencias válidas del nodo : ABI al nodo ABCG
false
Chequear referencias válidas del nodo : ABI al nodo ABEH
false
Chequear referencias válidas del nodo : ABI al nodo AB
false

Chequear referencias válidas del nodo : ABCJ al nodo ABCG
false
Chequear referencias válidas del nodo : ABCJ al nodo AF
false
Chequear referencias válidas del nodo : ABCJ al nodo ABI
false

Chequear referencias válidas del nodo : ABIK al nodo ABCD
false
Chequear referencias válidas del nodo : ABIK al nodo ABEHL
false
Chequear referencias válidas del nodo : ABIK al nodo ABIK
true

Chequear referencias válidas del nodo : ABEHL al nodo ABCD
false
Chequear referencias válidas del nodo : ABEHL al nodo AB
false
Chequear referencias válidas del nodo : ABEHL al nodo HEAP
False
```

Texto 4. Chequeo referencias válidas mediante Write Barrier

3.4.- Otra visión de modelo

Modelo con referencias cíclicas permitidas

Con este modelo vamos a omitir la regla del único padre, con el fin de generar referencias cíclicas. Artículo de *M.T. Higuera Toledano* [4].

Proponemos un ejemplo para explicar el modelo; En este ejemplo van a coexistir dos áreas de memoria de tipo *Scoped*, *ScopedA* y *ScopedB* y 2 hilos de tiempo real η1 y η2., la situación estudiada es en la que el η1 ha ejecutado *las Scoped A* y la *ScopedB* y el η2 por su parte, ha ejecutado el método ***enter****()* para la *ScopedB* y la *ScopedC*

En un primer momento el hilo η1 ejecuta la región *ScopedB*; inmediatamente después η2 también intenta ejecutar el mismo área de memoria *ScopedB*. En este momento el hilo η1 ha establecido una relación con el área de memoria *ScopedA*, en la que *ScopedA* es padre del área de memoria *ScopedB*, mientras que para el η2 la relación de *ScopedB* en su hilo es que el es el primer padre existente. *(Figura 41).*

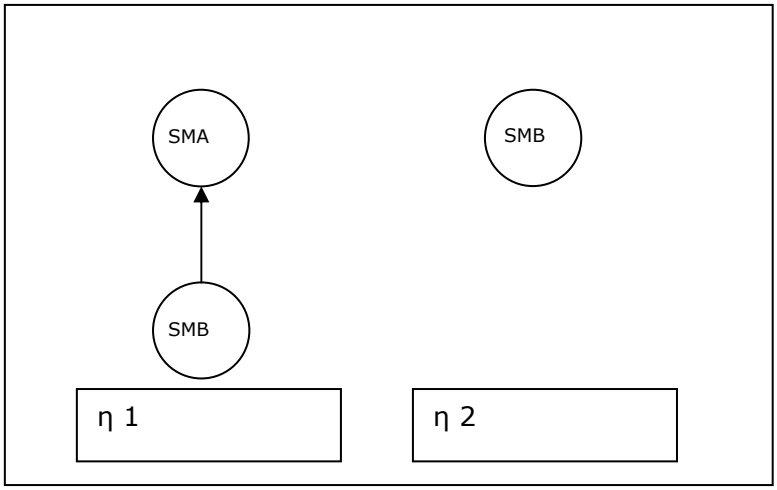


Figura 41. Ejecución hilos

Con este modelo se intenta eliminar la regla del único padre, estableciendo el conjunto de relaciones a la entrada de las distintas áreas de memoria en los hilos de ejecución. Se permitirán pues ciclos dentro de las referencias entre las distintas áreas de memoria, teniendo en cuenta que las *Scoped* que posean un ciclo serán recogidas al mismo tiempo.

Una solución para soportar esta implementación sería mediante el uso de un *array* de bits, poseyendo una posición por cada área de memoria de tipo *Scoped* en el sistema, donde cada entrada poseerá una marca con la *Scoped*

que deberá ser recolectada previa a ella misma y un número entero indicando el número de referencias hechas a esa región de memoria. Por ejemplo si un hilo de ejecución de tiempo real ejecutase el método ***enter***() primero para una *ScopedA*, luego para una *ScopedB* y más tarde de nuevo para la *ScopedA*, se generará un *array* como aparece en la *Figura 42*.

	count	scopedA	scopedB
scopedA	2		X
scopedB	1	X	

Figura 42. Ejecución método enter()

Como se puede observar el contador de referencias de la *ScopedA* está a dos ya que han sido 2 veces las que se ha ejecutado el método ***enter****()* para esa región de memoria, teniendo como área de memoria de recolección previa a la *ScopedB*, ya que ésta entró tras ella.

Pero hay que tener en cuenta que debido a los ciclos dentro de esta implementación en la zona de información de la *ScopedB* el contador de referencias está a uno, ya que sólo ha entrado una vez, pero la marca de la región de memoria que deberá ser recolectada previamente está a la región *ScopedA*. Se está produciendo un ciclo dentro de as referencias de parentesco. Figura 43

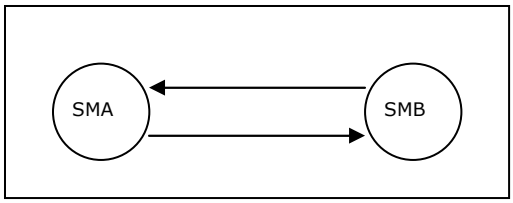


Figura 43. Ciclos Scoped

Debido a los contadores de referencias de las áreas de memoria y a las marcas de recolección previa, se deberá recolectar primero la región *ScopedA* para más tarde hacer lo propio con la B y acabar con la A, para realizar esto será necesario modificar las interacciones de la RTSJ con los contadores de referencias de las regiones de memoria de tipo *Scoped* ya que en este caso el contador de referencias para el área ha llegado a 0 como se ve en la Figura 44.

	count	scopedA	scopedB
scopedA	1		X
scopedB	0	X	

Figura 44. Recolección Scoped A

En la implementación típica RTSJ y para este caso, cuando el contador de referencias de un área de memoria llegase a 0, esa área sería recogida y su posición en el *array* eliminada. Esta naturaleza del algoritmo ha de ser modificado para que una vez que se lleve a 0 en el contador de referencias de un área de memoria determinado hará falta recorrer el conjunto de marcar de recolección previa de áreas de memoria, para cerciorarse que puede ser eliminado sin causar una violación del modelo de recolección de regiones, por lo que mientras no se eliminen todas las marcas de su fila no podrá ser eliminado ni por lo tanto recogido. El algoritmo de recolección se ve en el algoritmo *CollectingRegion*(i).

```
Para cada región j perteneciente a las regiones que se colectan antes de i

        Si el contador de referencias de j es 0
                Desacoplar(i,j)
                Si la región de memoria previa de colectar de i es null
                        Recolectar región j
                Fin Si
        Fin Si
Fin Para

Si la región de memoria previa de colectar de i es null
        Hacer recolección de i
Fin Si
```

Algoritmo 7. CollectingRegion

La ejecución del algoritmo de recolección ejecutado para la región *ScopedB* en el punto de la *Figura 19* no tendrá efecto, ya que el contador de referencias del área de memoria que será en el área de memoria que debe ser recolectado previa a él es aún mayor que 0. Por otro lado la ejecución del algoritmo con la *ScopedA* producirá el desmarcado de la *ScopedB* como región previa recolectada de su *array*, ya que su contador de referencias está a 0; una vez hecho esto el contador de referencias de la región A bajará a 0, lo que permitirá que ejecutando una vez más el algoritmo en la *ScopedB* elimine la marca de región de recolección previa. Esta situación aparece representada en la Figura 45.

	count	scopedA	scopedB
scopedA	0		
scopedB	0	X	

Figura 45. Recolección Scoped B

Un problema de este modelo es el uso de memoria, ya que las regiones permanecerán vivas aun cuando su contador de referencias haya llegado a 0, ya que deberán ser comprobadas en pasos posteriores sus referencias a las demás áreas de memoria en cuanto a la recolección previa.

3.5.- Manual de usuario

En este apartado vamos a mostrar un conjunto de ejemplos de nuestra aplicación de simulación para cada modelo planteado en las secciones anteriores. Intentaremos que sean lo más intuitivo y explicativo posible de forma que se entienda de forma sencilla y sin mucho trabajo lo que hemos intentado hacer entender por medio de este desarrollo. Sólo mostraremos las pruebas relativas a los dos modelos implementados bajo la naturaleza de varios hilos de ejecución.

3.5.1.- Descripción de la interfaz

La interfaz será común para ambos modelos, cambiando por otra parte el contenido de las imágenes. Esta interfaz constará de dos paneles que llamaremos pare diferenciarlos durante toda esta sección de pruebas como panel A, refiriéndonos al de la izquierda y panel B al de la derecha (Figura 46).

Este panel va a contener dos botones situados en las dos esquinas de la parte inferior de la pantalla:

- Introduce Hilo: situado en la parte inferior izquierda marcado con una elipse, encargado de introducir en el programa con cada pulsación un nuevo hilo. Cada nuevo hilo se irá nombrando con un entero, empezando desde el número 1 y aumentando sucesivamente en una unidad.

- Mata Hilo: botón de la parte inferior derecha redondeado con una elipse. Una vez se han ido introduciendo hilos en el programa, la acción de este botón producirá que el recolector actúe sobre el hilo marcado en la lista.

Por otro lado existirá un combobox marcado por un rectángulo de líneas discontinuas. Esta lista se irá rellenando por el conjunto de hilos introducidos por el usuario, pulsando encima de ella una ventana se desplegará mostrando

todos los hilos vivos en ese momento. Su función principal es para seleccionar el hilo que se desee eliminar del árbol mediante el botón de matar el hilo

Hay que tener en cuenta para todas las pruebas que vamos a mostrar que usaremos los parámetros usados como hemos ido comentado a lo largo de la memoria, es decir, cada hilo tendrá diez acciones asociadas, pudiendo ser cada una la inserción de un nuevo Memory Area siendo el tipo elegido aleatoriamente, teniendo cada clase la misma probabilidad de ser seleccionada; o por otro lado la acción elegida la de referenciación a tres Memory Areas elegidas también de forma aleatoria.

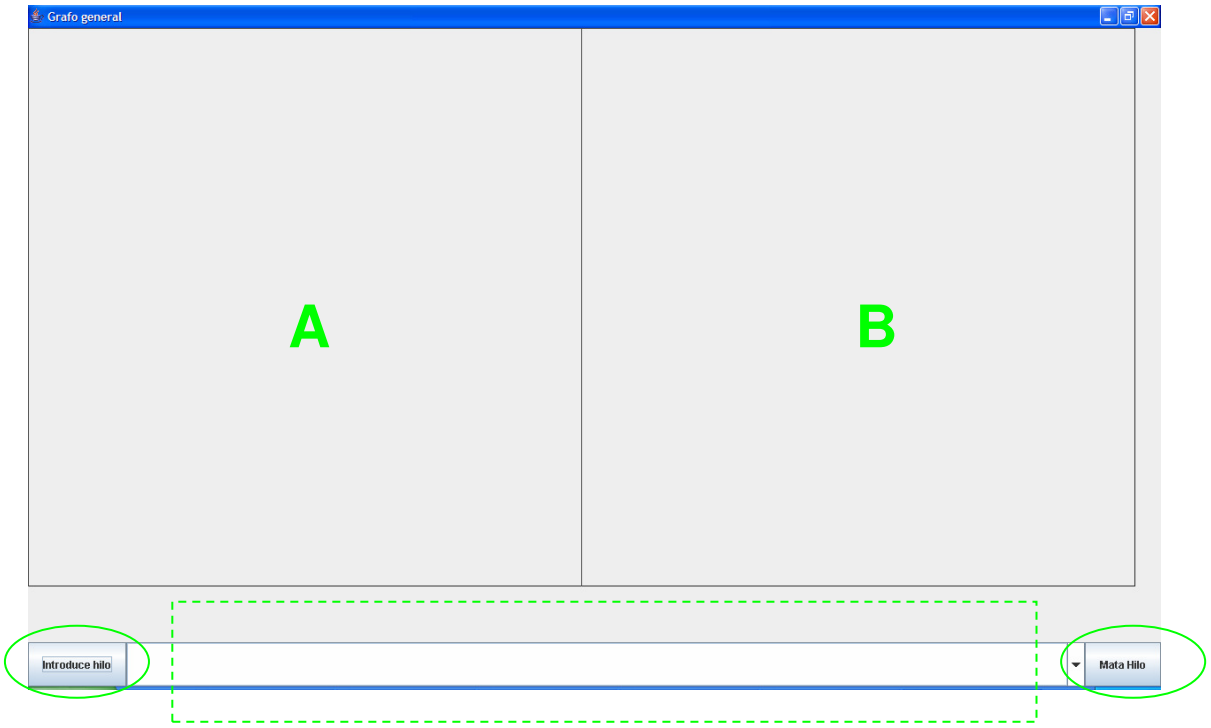


Figura 46. Interfaz inicial

3.5.2.- Modelo de hilos

Este modelo ha sido denominado más arriba como solución al modelo; es como hemos comentado antes la variación con respecto al modelo inicial introduciendo el concepto de hilos de ejecución.

Durante el proceso de la muestra de las pruebas las distintas Scoped Memory van a tener una estructura similar a la figura x, de tal forma que sabremos, como hemos comentado antes que se trata de un Memory Area de tipo Scoped debido a sus iniciales SM, que el nombre y por tanto el orden en que ha sido entrado es 4, que posee 3 referencias a su área de memoria y que le han entrado 3 hilos, los denominados 2,4 y 5 (Figura 46).

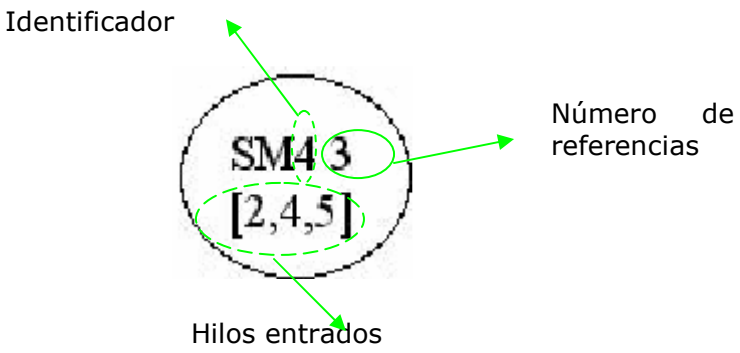


Figura 47. Nodo hilos

En principio tras la ejecución de este modelo se presentará la interfaz comentada en la Figura 46. Para realizar la prueba de este modelo vamos a realizar una serie de pasos comentando en cada uno los que realizamos en relación a la interfaz y los resultados gráficos obtenidos, mostrando en cada caso las posibles acciones internas que han ido sucediendo.

Paso primero, inserción del primer hilo. (Figura 48)Pulsando en el botón de introduce hilo hemos logrado crear un nuevo hilo, denominado en este caso como aparece en el combobox de abajo *Hilo de ejecución numero 1.*

Como se puede observar se han producido 7 inserciones en el árbol de Memory Areas en A, manteniendo en todo caso la regla del único padre en todas las Scoped Memory existentes en A.

Por otro lado en B se muestra el subárbol de Scoped Memory, en que ya sólo aparecen las áreas de memoria de este tipo, manteniendo en todo momento

las mismas relaciones de parentesco que se producían en el árbol general de A.

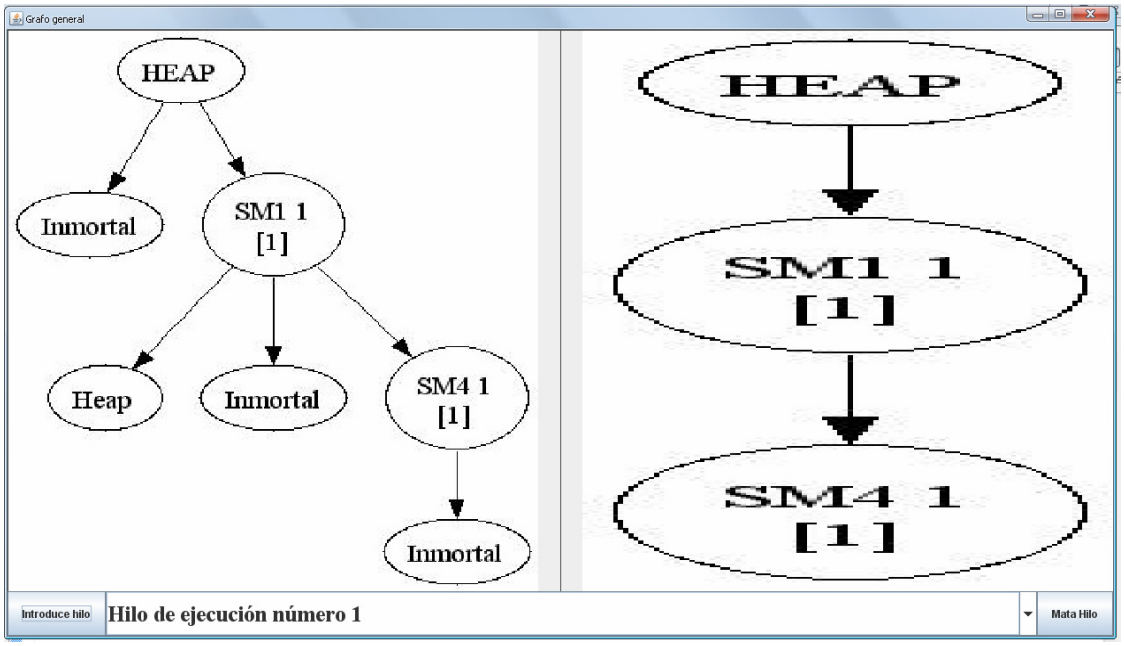


Figura 48. Primera inserción primer modelo

Paso segundo, vamos a introducir 3 nuevos hilos, de tal forma que en el combobox aparecerán los 3 nuevos hilos con sus respectivos identificadores y ambos árboles sufrirán sendas modificaciones en función de las nuevas relaciones de parentesco creadas.

Pulsando en el botón del combobox se desplegará la lista de hilos existentes en este momento. Figura 49

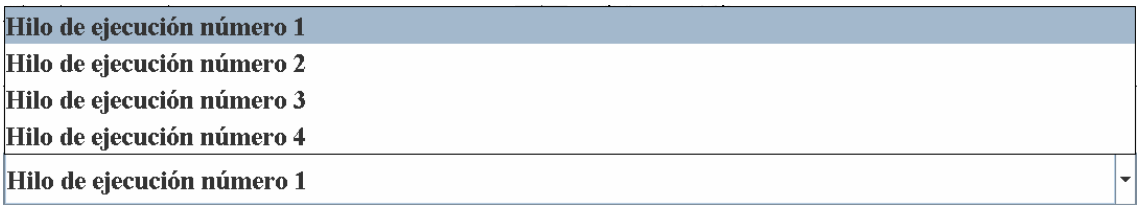


Figura 49. Combobox ejemplo

Eligiendo con otra pulsación de botón sobre el hilo deseado éste quedará como marcado para poder proceder, por ejemplo a su eliminado del árbol.

Tras la inserción de tres nuevos hilos en ambos árboles se ha producido los siguientes cambios en la interfaz (Figura 50):

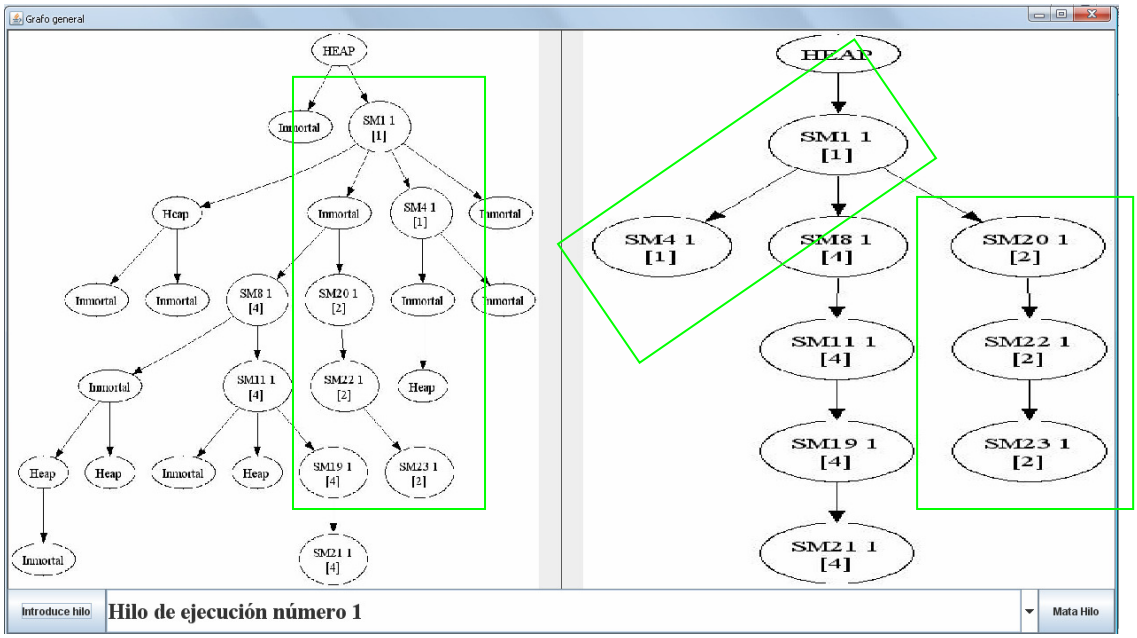


Figura 50. 4 Hilos de ejecución

Se observa el crecimiento del árbol debido a las nuevas inserciones de Memory Areas y referencias de cada hilo a las ya creadas.

Vamos a centrar una imagen en la parte recuadrada de la figura 51 en su lado A:

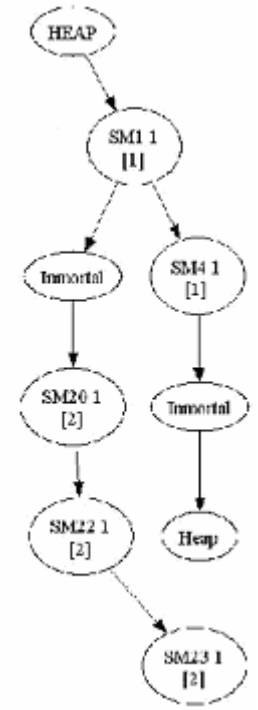


Figura 51. Parte Árbol

El hilo de ejecución 2 va a estar representado en esta imagen, ya que se ve que estará dentro de las Scoped Memory 20, 22 y 23, como se observa en su las listas de hilos de cada una.

Esta parte del árbol de Memory Areas va a generar la parte del árbol recuadrado en la parte B de la interfaz, estando acercada esta imagen en la Figura 52.

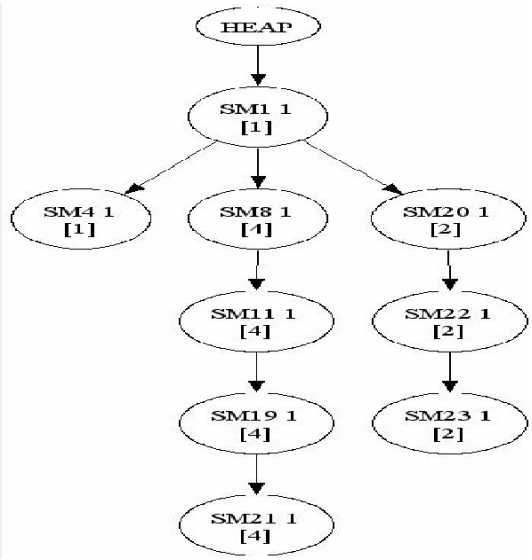


Figura 52. Parte subárbol

Las relaciones existentes entre las Scoped Memory del árbol general se mantienen en el subárbol de Scoped, y en todo momento se mantiene la regla del único padre.

Paso tercero eliminación de un hilo de ejecución. Mediante la selección de, por ejemplo el hilo 1 de ejecución, y el posterior pulsado del botón de matar hilo se procederá a la eliminación de este hilo de ejecución, esto va a conllevar la eliminación de todas aquellas Scoped Memory que sólo posean el hilo de ejecución 1, y la eliminación de este hilo de las listas del resto que la posean, generando la interfaz de la Figura 53.

Centrémonos ahora en la parte recuadrada de la figura de B.

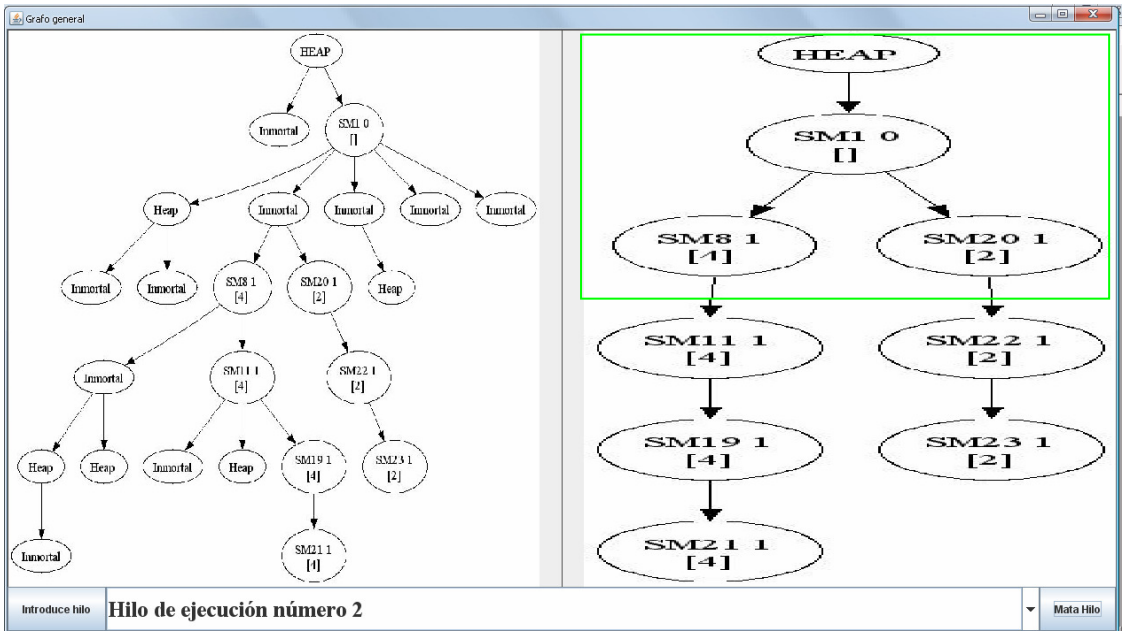


Figura 53. Muerte hilo 1

Como se ve en la Figura 54, la Scoped Memory 1 ya no posee ningún hilo de ejecución en su lista de hilos, pero no ha sido eliminada del árbol. El motivo de esto es que no puede ser eliminado ya que de ella cuelgan otras Scoped Memory que aún están vivas(en ejecución) y por lo tanto deberá esperar a que están terminen para poder liberar su memoria.

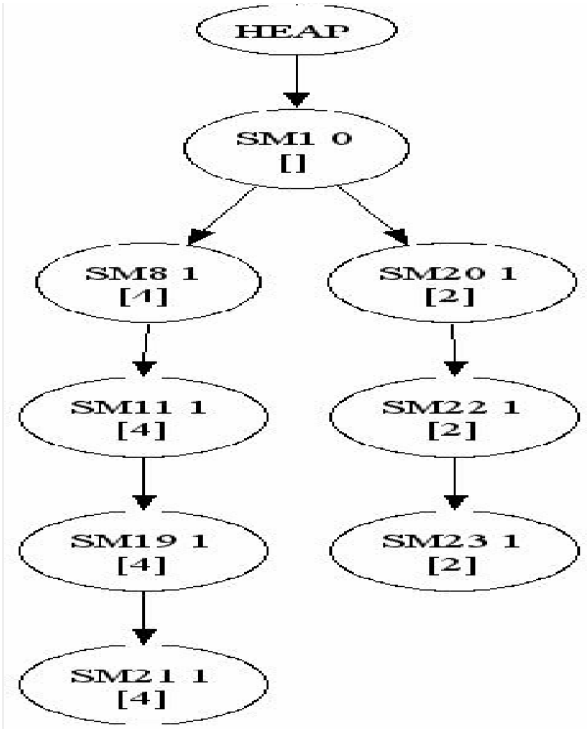


Figura 54. Parte árbol muerte del hilo 2

El siguiente paso para poder ver que ocurre esto va a ser matar el hilo 4 y luego el hilo 2, que una vez terminados implicarán la recolección tanto de ellos como de los Scoped Memory que ya puedan ser recolectados Figura 55.

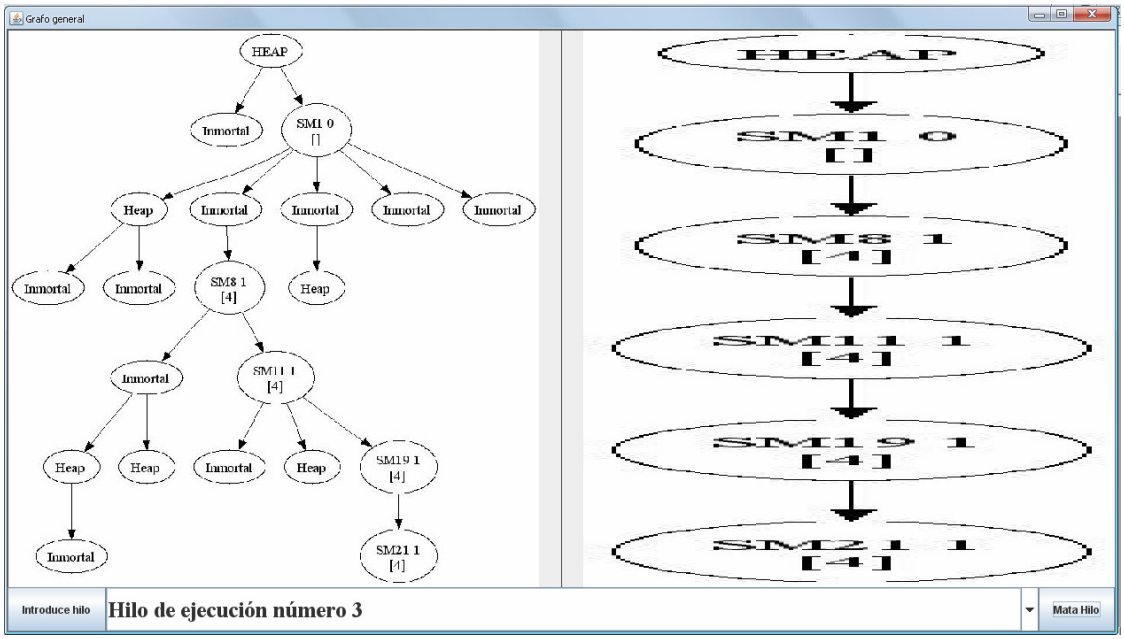


Figura 55. Muerte hilo 2

Cada vez que vamos eliminando hilos del programa se va recortando el árbol de la parte B en gran medida, mientras que en el A como existen las áreas de Memoria Inmortal y Heap sólo afectarán las eliminaciones de los hilos a las eliminaciones de las Scoped Memory, aunque si deberán mostrarse en todo momento los cambios en las relaciones debido a esas supresiones.

En este momento van a quedar vivos los hilos 3 y 4.

Si eliminamos el hilo 3 el grafo quedará de la misma manera porque no ha generado ninguna región de memoria de tipo ScopeMemory (Figura 56):

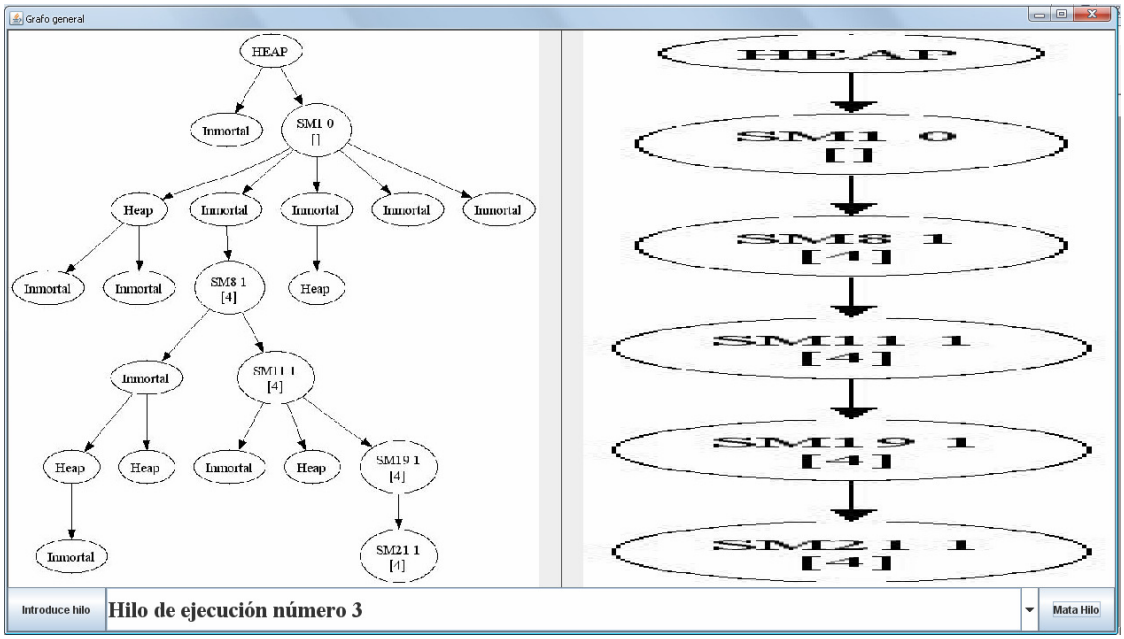


Figura 56. Hilo vivo 3

Eliminando el último hilo vivo, el 4, vamos a observar que en la parte A, van a quedar únicamente las relaciones que quedan de los tipos Heap e Inmortal, mientras que en la parte B el subárbol de Scoped ha desaparecido completamente , habiéndose producido la recolección completa eficientemente Figura 57.

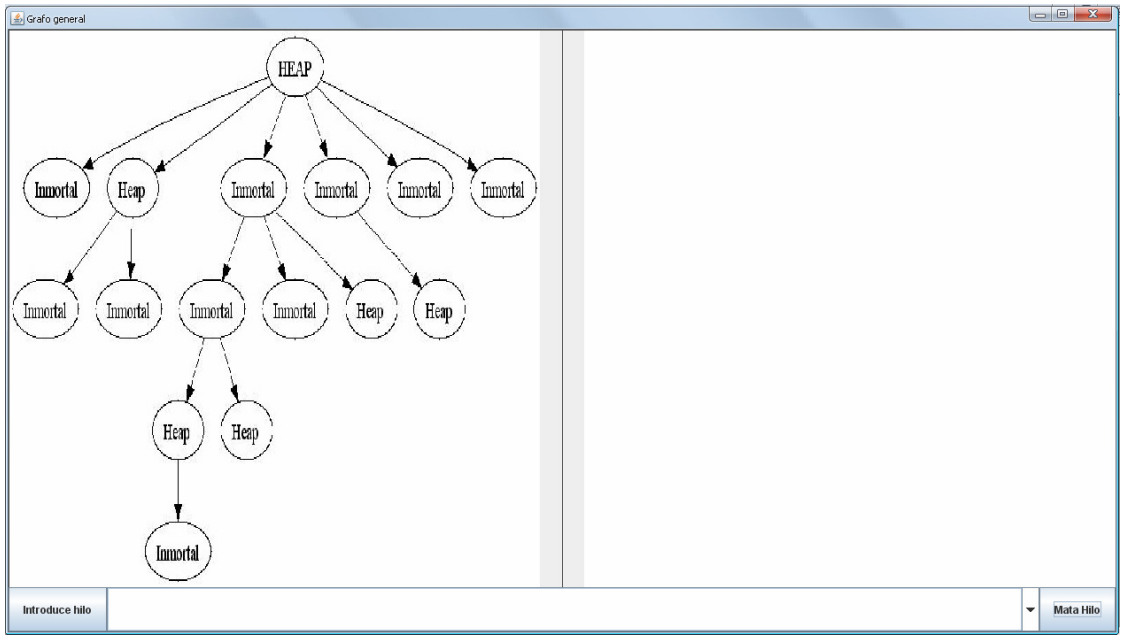


Figura 57. Ultima interfaz hilos

3.5.3.- Modelo de nombres

A continuación vamos a relatar la pruebas referentes al modelo denominado más arriba como la solución alternativa, es decir, la basada en nombres. En este caso no vamos a mostrar en ningún momento las regiones de memoria de tipo Heap e Inmortal, por tanto todos los árboles representados en la interfaz van a ser de tipo Scoped Memory.

La forma de estructurar la interfaz será de tal forma que en la parte A de la interfaz se mostrará el árbol antiguo, es decir el de antes de una inserción o eliminado de un hilo, mientras que en la parte B mostraremos el árbol de Scoped con la nueva estructura tras la inserción o supresión del hilo.

Los nodos serán como hemos dicho de naturaleza Scoped y su estructura será la siguiente (Figura 58)

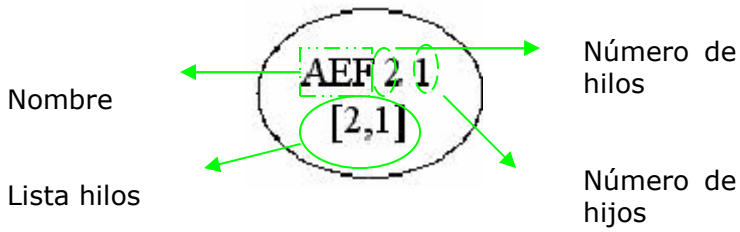


Figura 58. Nodo nombres

Cada nodo tendrá reflejado por tanto esas características en todo momento, el nombre del Scoped, el número de hijos que tiene, el número de hilos que han entrado en el y el conjunto de estos hilos.

Vamos a proceder a explicar como hemos hecho antes una serie de pasos de creación e ínter actuación de hilos y cómo se refleja cada acción en la interfaz. La apariencia inicial de la interfaz es la comentada más arriba en la figura SDSD.

El primer paso a realizar como hemos hecho arriba será la inserción de un primer hilo de ejecución mediante el pulsado del botón introduce hilo, esta acción insertará en la lista de combobox el hilo de ejecución número 1 y producirá la siguiente estructura de la interfaz (Figura 59):

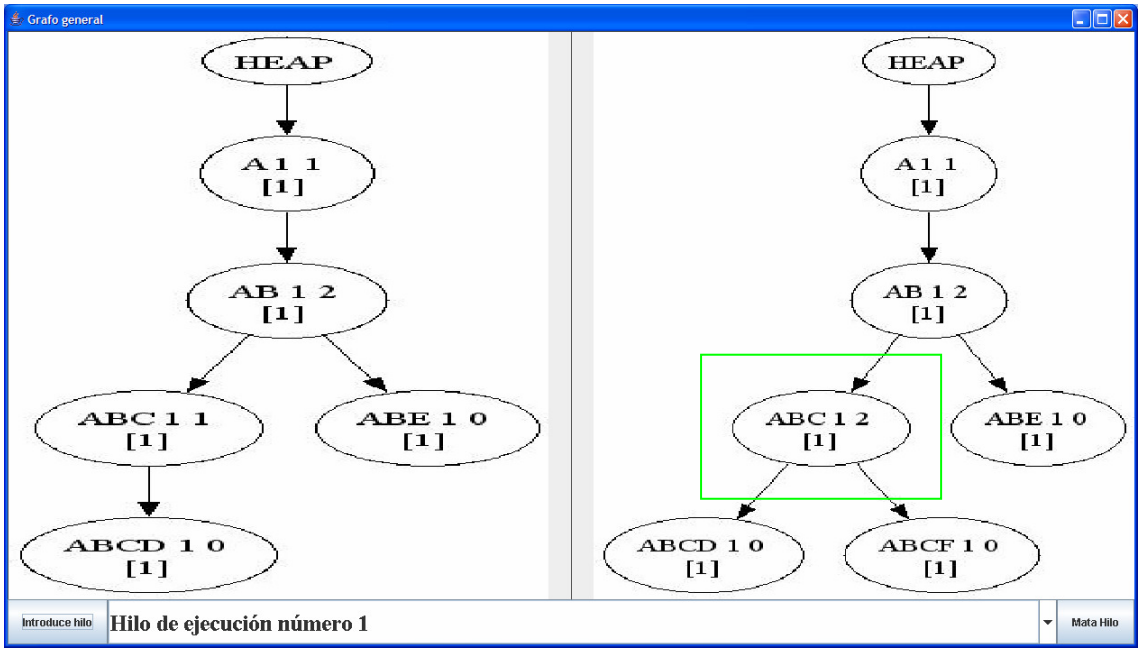


Figura 59. Inserción hilo 1 nombres

Como podemos observar en la parte A está el árbol de Scoped sin la última acción realizada, la inserción del nodo ABCF. Como se puede ver la regla del único padre de este modelo se cumple en todo momento y en cada parte del árbol. Centrando la imagen en la imagen recuadrada podemos observar que el contadores de hilos e hijos es correcto, ya que obviamente sólo existirá un hilo, y el contador de hijos es 2, reflejando sus dos hijos ABCD y ABCF; por otro lado la lista de hijos de este nodo sólo tendrá el único hilo en ejecución en este momento, el 1.

Se puede ver que el thread 1 que está en ABC, ha creado dos regiones hijas, la ABCD y ABCF. Luego entra en ABCD, hace un executeInArea() sobre AC y un ABCF.enter().

Cuando termina el método ACF.enter() la región activa es AC y cuando termina el método ABCD.executeInArea() vuelve ABCD a ser la región activa.

En este caso vamos a introducir de forma seguida dos nuevos hilos de ejecución manteniendo vivo el primer hilo introducido, por tanto coexistirán en ejecución 3 hilos, el 1, 2 y 3, generando los árboles antiguo y nuevo en las partes A y B respectivamente (Figura 60):

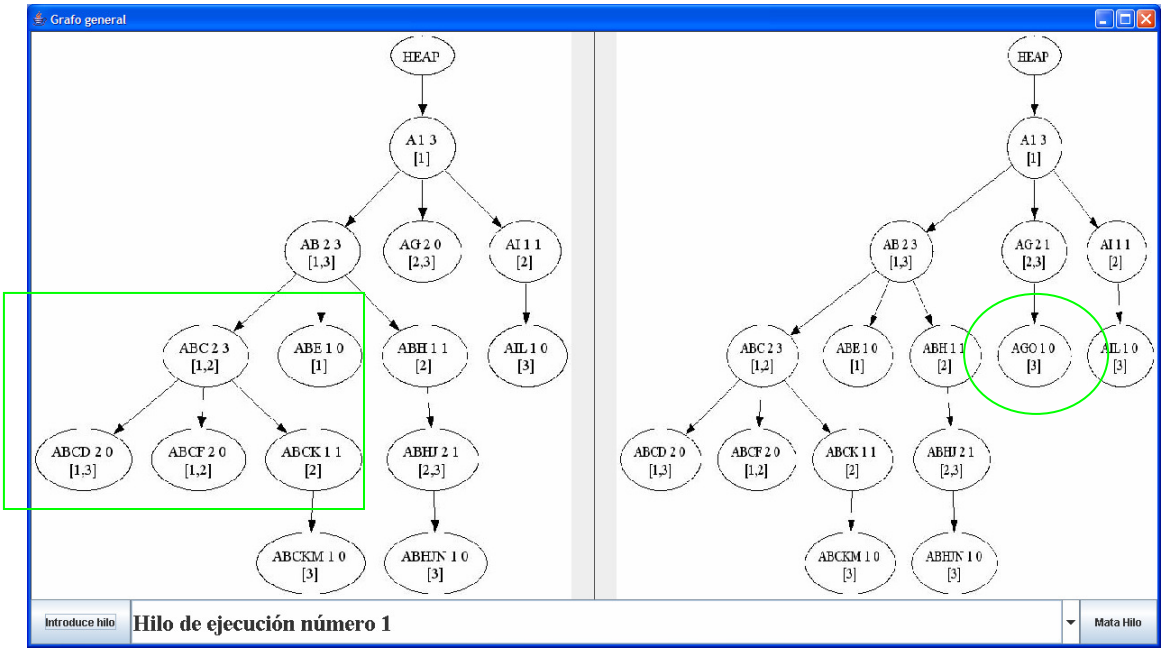


Figura 60. Tres hilos en nombres

En este caso la diferencia existente entre las dos situaciones reside en la creación del nodo AGO marcado mediante la elipse.

Vamos a centrar la imagen en la parte recuadrada para ver que situación podemos apreciar en esa parte (figura 61):

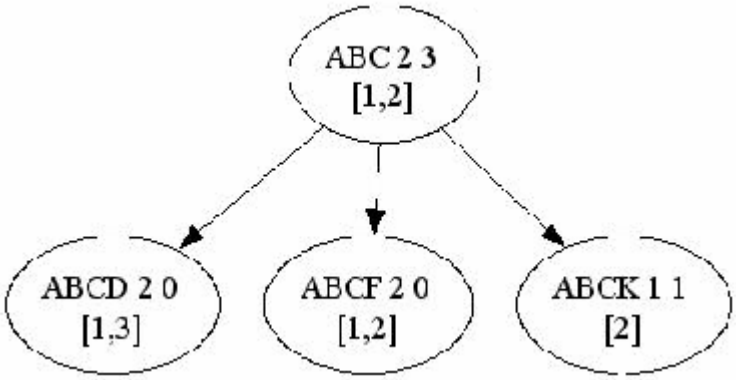


Figura 61. Parte árbol nombres

El área de memoria Scoped ABC va a poseer como indica sus atributo de numero de hijos 3: ABCD, ABCF y ABCK; poseerá 2 hilos en ejecución dentro de él que serán los nombrados en su lista de hilos: el 1 y el 2. Por otro lado el ABCD al ser hoja no tendrá ningún hijo pero si tendrá dos hilos entrados en su área de memoria, el 1 y el 3. La Scoped Memory ABCK sólo tendrá un hilo dentro de él, el 2, pero tendrá un hijo el ABCKM, viéndolo en la figura 60.

El siguiente paso que vamos a realizar para explicar el funcionamiento de la implementación de hilos basado en nombres será, por ejemplo, la eliminación del hilo de ejecución 1. Esto va a producir la siguiente pantalla (Figura 62):

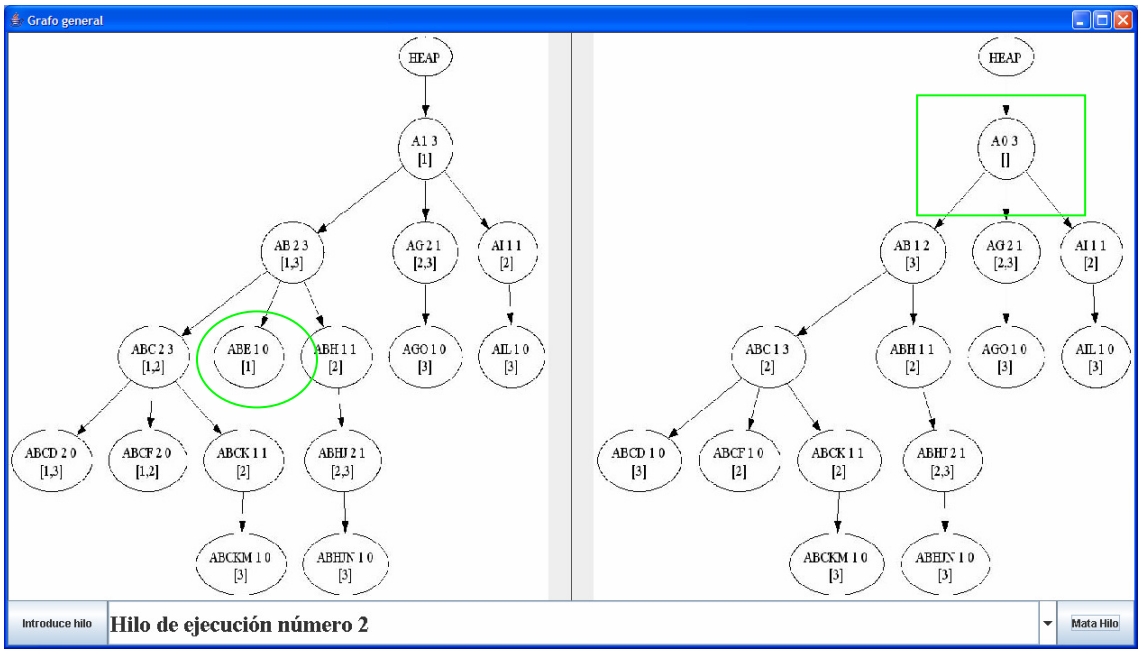


Figura 62: Eliminación hilo 1 nombres

Como se puede observar el árbol de la parte A de la Figura 61, es la misma que la parte B de la Figura 59, ya que la última acción tras la eliminación será la nueva antes de esa acción.

Tras la eliminación del hilo uno se aprecia que se ha recogido la Scoped Memory marcada con la elipse, ya que al ser hoja y sólo poseer el hilo 1 dentro de ella ha podido ser recogida por parte del recolector. Más arriba vemos que ocurre el mismo fenómeno que ocurría antes; en la parte recuadrada (Figura 63):

Figura 63. Scoped sin hilos nombres

Esta Scoped que como indican sus atributos no posee ningún hilo dentro, pero como sí que tiene 3 hijos no podrá ser recolectado hasta que los 3 hijos lo sean previamente.

La siguiente acción a estudiar va a ser la eliminación del hilo de ejecución 3, produciéndose una situación como se muestra en la Figura 64.

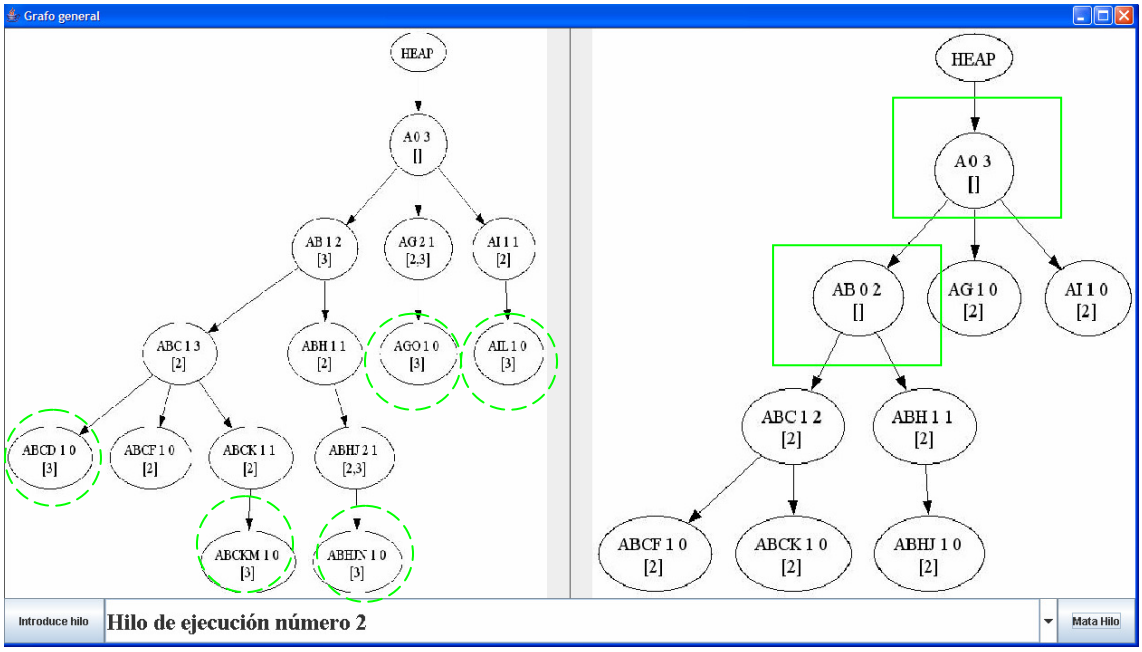


Figura 64: Eliminación hilo 3 nombres

Como ocurría antes se han eliminado aquellos Scoped Memory que cumplían las condiciones de ser recolectadas, mientras que aquellas que no permanecen hasta que sus hijos sean recolectados. La situación de que una Scoped Memory no posee ningún hilo dentro de él pero aun tiene que esperar a que sus hijos salgan se ve en las partes recuadradas de la parte B.

La otra situación, en que una Scoped posee el contador de hilos y el número de hijos a 0 y por tanto podrá ser recolectada se ve en las situaciones marcadas con elipses en la parte A.

Quedará pues en ejecución el hilo 2, que tras eliminarlo pulsando como hemos hecho en todas las veces anteriores el botón de Matar Hilo quedará el árbol nuevo totalmente vacío (Figura 65), lo que indicará que han sido recolectadas todas las Memory Areas eficientemente, teniendo, como en los casos anteriores la situación antigua de la parte A la que era nueva de la figura 64.

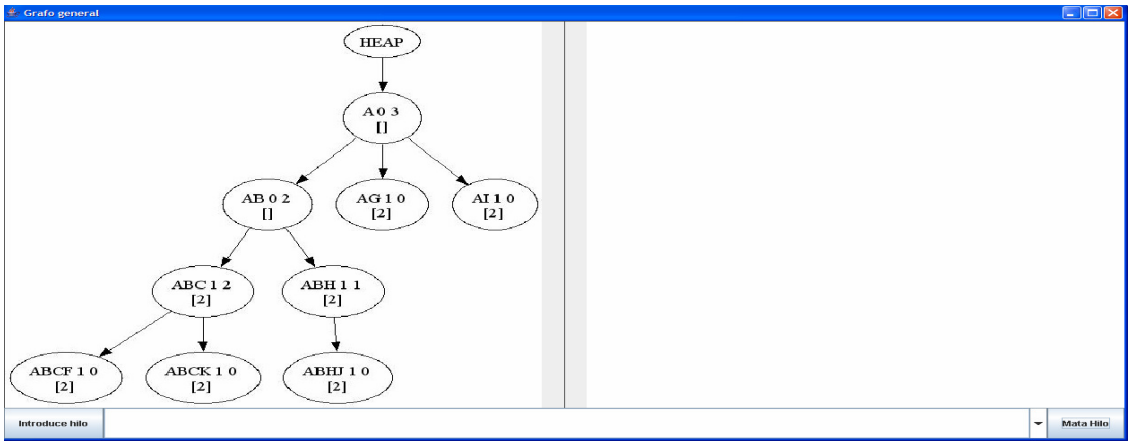


Figura 65. Árbol vació de nombres

4.- Referencias

[1] Angelo Corsaro and Ron K. Cytron. Efficient Memory-Reference Checks for Real-time Java. Department of Computer Science and Engineering, Washington University

[2] M. Teresa Higuera-Toledano. The indeterministic Behavior of Scoped Memory in Real-Time Java

[3] M. Teresa Higuera-Toledano.Towards an Understanding of the Behavior of the Single Parent Rule in the RTSJ Scoped Memory Model

[4] M. Teresa Higuera Toledano. Allowing Cycles References among Scoped Memory Areas in the Real-Time Specification of Java

[5] The Real-Time Specification for Java. http://www.rtsj.org

[6] M. Teresa Higuera-Toledano and Valerie Issarny. Java Embedded Real-Time Systems: an Overview of Existing Solutions.

[7] Angelo Corsaro and Ron K. Cytron. Implementing and Optimizing Real-Time Java

[8] Mike Rainey. Exploring Real-Time Java

[9] Peter C. Dibble. Real-Time Java. Platform Programming

[10] http://www.ovmj.org

[11] http://www.ovmj.org/mailman/listinfo/ovm

[12] http://www.ovmj.org/mantis

[13] http://www.arrakis.es/abelp/ApuntesJava/Threads.htm

[14] http://www.cs.york.ac.uk

[15] http://jrate.sourceforge.net

[16] http://www.rtsj.org

[17] http://www.wikipedia.org

5.- Apéndice

En las primeras fechas del curso hubo un intento de instalación de varias máquinas virtuales en un entorno Linux para su posterior optimización.

Para ello nos bajamos de varias páginas oficiales los distintos compiladores y máquinas virtuales java. La mayoría de máquinas daban distintos errores, algunos subsanados y otros como a más gente le había pasado (visto por la web) imposibles de recuperar.

Las dos configuraciones de maquinas virtuales probadas con más entusiasmo fueron jRate y OVM.

5.1.- jRate

5.1.1.- Características

Es un GCJ (Compilador de GNU para Java) con las siguientes caracteísticas principales:

• "ahead-of-time compiler". Puede compilar:
- Código fuente Java a código máquina
- Código fuente Java a "bytecode" (ficheros *.class)
- "bytecode" a código máquina

• Las aplicaciones se enlazan con la librería de tiempo de ejecución (libgcj)

jRate (Java Real-Time Extension)

• Extensión del compilador GCJ y de su librería de tiempo de Ejecución

• Añade mucha de la funcionalidad definida en la RTSJ

• Escrita en C++ y Java

Por otro lado en cuanto a entorno de jRate se puede decir que es de desarrollo complejo. Entre otras características hemos encontrado:

• compilación completa del GCC (es delicado hacer cambios) jRate se basa en LinuxThreads (interfaz POSIX)

• adaptación: compilar con los ficheros '*.h' de MaRTE OS

Problemas encontrados para jRate:

• Versiones de compiladores y utilidades

• Enlazado: librerías, versiones, orden de las librerías

•jRate no comprueba valor de retorno de las funciones POSIX

•Constructores de objetos estáticos de C++ se ejecutan antes de la inicialización de MaRTE OS

jRate ha sido realizada en el grupo RTS de York (Prof. Andy Wellings)

5.1.2.- JRate ¿Cómo instalarlo?

Para instalar jRate necesitamos tener previamente instalado gcc-3.3.3 y jre-1.5.0_09-linux-i586.

Instalación de jre:

- Nos descargamos de Sun el archivo necesario: jre-1_5_0_09-linux-i586-rpm.bin**.**
- Nos ponemos como super usuarios mediante el comando '*su'* de linux e introduciendo la clave de *root*.
- Una vez hecho esto cambiamos el permiso del archivo descargado para que sea ejecutable:

  chmod a+x jre-1_5_0_09-linux-i568-rpm.bin

- Empezar el proceso de instalación:

  ./jre-1_5_0_09-linux-i568-rpm.bin

  Se mostrará el contrato de licencia de archivos binarios, al llegar al final escribimos SI para proseguir con la instalación.
  Una vez finalizada la instalación se mostrará la palabra *terminado*.
  El archivo de instalación crea un archivo con extensión .rpm.

- Run del comando RPM en el terminal para instalar los paquetes:

  rpm –iv jre-1_5_0_09-linux-i568.rpm

- Configuración de las variables de entorno:

  Tenemos que editar el archivo .bash_profile del usuario, por ejemplo en nuestro caso : /home/raquel/.bash_profile, siempre como super usuario, y editamos:
  JAVA_HOME = /usr/var/jre1.5.0_09
  PATH = $PATH:$HOME/bin:$JAVA_HOME/bin
  export PATH JAVA_HOME

- Para comprobar la instalación escribimos en la línea de comandos:

  java –versión, esto ha de mostrarnos la versión del **jre** que hemos instalado.

Instalación de gcc:

- Nos descargamos los archivos necesarios :

  gcc-core-3.3.3.tar.tar
  gcc-java-3.3.3.tar.tar
  gcc-g++-3.3.3.tar.tar

- Escribimos en la consola de comandos:

  tar xfj gcc-core-3.3.3.tar.tar
  tar xfj gcc-java-3.3.3.tar.tar
  tar xfj gcc-g++-3.3.3.tar.tar

  Con esto lo que hacemos es descomprimir los archivos que nos hemos descagado.

- Escribimos en la consola de comandos la orden *ls* y nos debe aparecer gcc-3.3.3.
  Una vez hecho esto escribimos:

  mv gcc-3.3.3 gcc
  ./configure
  make

  Y con esto queda instalado gcc.

Una vez que hemos instalado lo que necesitamos ya podemos empezar con el proceso de instalación de jRate:

- Descomprimimos el archivo *jrate* que nos hemos descargado.
- Configuramos las variables de entorno. Para ello nos metemos en el directorio creado al descomprimir el archivo, y dentro de éste en el directorio /script. Aquí creamos un archivo en el que vamos a editar las variables de entorno mediante los siguientes comandos:

  emacs jRate-env.sh &

  Ahora en jRate-env.sh ponemos:

  export JRATE_SUITE_HOME=/home/raquel/Proyecto (directorio en el que tenemos descomprimido jrate)
  export JRATE_HOME=/home/Raquel/Proyecto/jrate-0.3.7.2-3.3.3
  export JRATE_GCC_SRC_HOME=/home/raquel/Proyecto/gcc

  Guardamos el archive y ejecutamos los comandos:

  ./jRate-env.sh
  source jRate-env.sh

  Para comprobar si las variables de entorno están bien ejecutamos los comandos:

  echo $JRATE_HOME
  que nos tiene que devolver /home/raquel/Proyecto/jrate-0.3.7.2-3.3.3
  echo $JRATE_GCC_SRC_HOME

que nos tiene que devolver /home/raquel/Proyecto/gcc

- Para construir jRate volvemos la directorio jrate mediante el comando *cd .. .*

Ejecutamos en la línea de comando la orden: ./configure

[raquel@localhost jrate-0.3.7.2-3.3.3]$ ./configure

Al final del proceso nos sale por pantalla lo siguiente:

```
================================================================================
jRate 0.3.7.2-3.3.3 is now configured.  Just type 'make'.
jRate version : 0.3.7.2
GCC version   : 3.3.3

If you are building for another target (like e.g. an embedded target), you
might want to review the system information I gathered in the file
jRate-config.h to ensure that it is correct for your target.  In particular,
the settings JRATE_NCPU, JRATE_PAGE_SIZE, JRATE_THREADMAX, CLOCK_FREQUENCY,
and the four default memory area sizes may need attention.  All of these
settings are user-specifiable with arguments to this configure script and,
for now, you MUST specify them to this script rather than just changing the
values in jRate-config.h directly.
================================================================================
If you don't unpack GCC sources into this directory before building:

/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3/gcc

then the proper version of the GCC sources (3.3.3) will be retrieved
from CVS (tag gcc_3_3_3_release) for you.
================================================================================
```

Copiamos la carpeta gcc (creada al instalar gcc 3.3.3) en el directorio jrate, volvemos a ejecutar la orden:

./configure

```
[raquel@localhost jrate-0.3.7.2-3.3.3]$ ./configure
```

Al final del proceso nos sale por pantalla lo siguiente:

```
================================================================================
===
jRate 0.3.7.2-3.3.3 is now configured.  Just type 'make'.
jRate version : 0.3.7.2
GCC version   : 3.3.3

If you are building for another target (like e.g. an embedded target), you
might want to review the system information I gathered in the file
jRate-config.h to ensure that it is correct for your target.  In particular,
the settings JRATE_NCPU, JRATE_PAGE_SIZE, JRATE_THREADMAX, CLOCK_FREQUENCY,
and the four default memory area sizes may need attention.  All of these
settings are user-specifiable with arguments to this configure script and,
for now, you MUST specify them to this script rather than just changing the
values in jRate-config.h directly.
================================================================================
===
```

jRate ya está configurado, ahora pasamos a construirlo mediante el comando:

make

El resultado es:

```
make de jrate
./.libs/libgcj.so: undefined reference to `_Jv_count_arguments(_Jv_Utf8Const*,
bool)'
./.libs/libgcj.so: undefined reference to `_Jv_JVMPI_Notify_THREAD_END'
./.libs/libgcj.so: undefined reference to
`_Jv_ResolvePoolEntry(java::lang::Class*, int)'
./.libs/libgcj.so: undefined reference to `_Jv_JVMPI_Notify_OBJECT_ALLOC'
./.libs/libgcj.so: undefined reference to `_Jv_JVMPI_Notify_THREAD_START'
collect2: ld returned 1 exit status
make[6]: *** [jv-convert] Error 1
make[6]: Leaving directory `/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3/gcc-
build/i686-pc-linux-gnu/libjava'
make[5]: *** [all-recursive] Error 1
make[5]: Leaving directory `/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3/gcc-
build/i686-pc-linux-gnu/libjava'
make[4]: *** [all-target-libjava] Error 2
make[4]: Leaving directory `/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3/gcc-build'
make[3]: *** [bootstrap] Error 2
make[3]: Leaving directory `/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3/gcc-build'
make[2]: *** [gcc.build-stamp] Error 2
make[2]: Leaving directory `/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory `/home/raquel/Proyecto/jrate-0.3.7.2-3.3.3'
make: *** [all] Error 2
```

Este error es el que nos sale siempre, a continuación comentamos algunas de las medidas que hemos tomado para intentar resolverlo:

- Consulta de manuales y artículos en Internet.
- Consulta de foros relacionados con la máquina virtual jRate.
- Hemos preguntado a varias personas familiarizadas con el entorno de Linux por los errores que nos iban saliendo y las soluciones que nos han dado no siempre han funcionado.
- Hemos instalado otras versiones de jre y jdk pensando que lo que fallaba era java, y nos han vuelto a salir los mismos errores.

Con lo cual llevamos atascados con este problema desde hace bastante tiempo.

Última solución adoptada → Probamos con otra máquina virtual: OVM

5.2.- OVM

5.2.1.- Características

OVM es una herramienta de generación de maquinas virtuales. Incluye varios compiladores, recolectores de basura y sistemas de generación y control de hilos de ejecución con el fin de mantener un sistema virtual. Genera eficientemente maquinas virtuales para programas sencillos en Java, y para programas escritos bajo las características de la especificación de tiempo real de Java

No está desarrollado por completo con el fin de mantener la especificación clásica de la maquina virtual de Java. Futuros avances serán:

- La carga dinámica de clases de distintas formas usando el compilador JIT
- El mantenimiento de referencias débiles
- El uso de APIs como AWT y otras sin núcleo, ya que actualmente depende de del JDK inicial, teniendo que generar un código propio para reemplazar las llamadas nativas

5.2.2.- OVM ¿Cómo instalarlo?

Para saber como se instala Ovm hemos tenido algunas dificultades porque la información encontrada ha sido escasa.

- Descargamos el archivo.
- Lo descomprimimos
- Y siguiendo un manual de instalación ejecutamos en la línea de comandos la siguiente instrucción:

./configure && make

```
[root@localhost OpenVM]# ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
checking for gawk... gawk
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for g++... g++
checking whether we are using the GNU C++ compiler... yes
checking whether g++ accepts -g... yes
checking for a BSD-compatible install... /usr/bin/install -c
checking for ranlib... ranlib
checking for ar... ar
checking for nm... nm
checking for makeinfo... makeinfo
checking for texi2dvi... texi2dvi
checking for install-info... no
checking for java... java
checking for javac... javac
checking for javadoc... javadoc
checking for gmake... gmake
checking Java property java.home... configure: error: Can't execute java program s with
java
```

Y nos sale este error que no sabemos resolver.

En un principio nos falló por otro motivo, no encontraba javadoc, pero instalando otra vez el jdk lo solventamos.

Como se ve con esta máquina no hemos podido hacer nada, ya que ni siquiera nos ha dejado configurar el archivo para posteriormente construirlo.

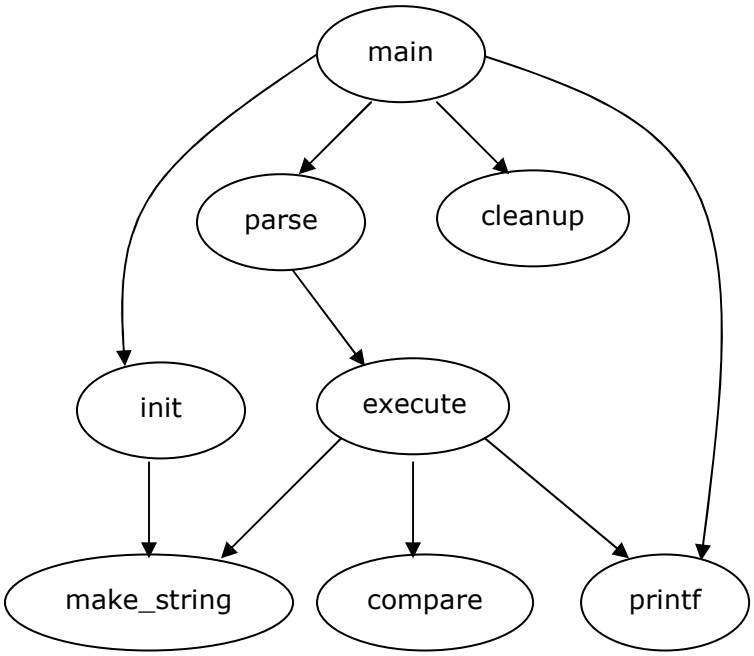
5.3.- Graphviz

5.3.1.- Características

Graphviz es una herramienta que nos facilita la generación de árboles de grafos. Es una manera de representar información estructural como diagramas de grafos abstractos. La creación automática de grafos toma gran importancia en aplicaciones de ingeniería del software, bases de datos y diseño web. Graphviz es software libre

Para que se generen correctamente los grafos de nodos debemos pasarle las instrucciones correspondientes en el lenguaje adecuado. Por ejemplo:

```
1: digraph G {
2:         main -> parse -> execute;
3:         main -> init;
4:         main -> cleanup;
5:         execute -> make_string;
6:         execute -> printf;
7:         init -> make_string;
8:         main -> printf;
9:         execute -> compare;
10: }
```



Con la primera instrucción decimos como se va a llamar nuestro grafo.
Los nodos que contiene y qué relación hay entre ellos lo especificamos con las siguientes instrucciones:

```
2: main -> parse -> execute;
3: main -> init;
4: main -> cleanup;
```

Con la instrucción 2 decimos que `main` va a ser el padre de `parse`, y que `parse` será el padre de `execute`.
En la 3 y en la 4 generamos dos nuevos hijos de `main`: `init` y `cleanup`, que pasarán a ser hermanos de `parse`.

Instalación

La instalación de Graphviz no tiene ninguna dificultad, simplemente hay que seguir las instrucciones. Su instalación es fundamental para la ejecución correcta del programa.

5.3.2.- Comunicación entre aplicación y Graphviz

Cada vez que termina un hilo en la aplicación y, por lo tanto, modifica el grafo del programa llamamos a Graphviz para que genere un grafo en un archivo .jpg y lo mostramos por pantalla. Para que Graphviz cree correctamente el grafo le pasamos un archivo .txt que, anteriormente, hemos generado al terminar cada hilo de ejecución.

Para generar el archivo .txt: recorremos todo el grafo viendo quien es el padre de todos los elementos (nodos) del grafo y lo insertamos en el archivo de texto en el lenguaje de Graphviz, como hemos visto arriba.

Para llamar al programa desde Java:

```
Process p = Runtime.getRuntime().exec("dot -Tjpg "+nombreArchivo+".dot
                                      -o "+nombreArchivo+".jpg);
```

"`nombreArchivo.dot`" dice a Graphviz cómo se llama el archivo de texto que hemos creado con todas las instrucciones necesarias para generar el grafo. La imagen de salida que genera Graphviz va a llamarse igual que el archivo de texto pero con extensión .jpg porque es una imagen.